A Prolog Garbage Collector for Aquarius

Hervé Touati
Computer Science Division
EECS
University of California
Berkeley, CA 94720

August 15, 1988

Abstract

This report presents the design and evaluation of the garbage collector we designed for the Aquarius project. Our design is the result of an attempt to incorporate into Prolog implementations the ideas which made generation scavenging successful for Lisp and Smalltalk. The main challenge was to take advantage of generation scavenging without giving away the basic Prolog technique of memory recovery upon backtracking based on stack deallocation. We were able to do so with little extra overhead at run-time. Our main strategy consists in restricting the action of the garbage collector to a *fixed* amount of memory allocated at the top of the global stack. This strategy has several advantages: it improves the locality of the executing program by keeping the data structures compacted and by allocating new objects in a fixed part of the address space; it improves the locality and the predictability of the garbage collection, which can concentrate its efforts on the fixed size area where new objects are allocated; and it allows us to use *simpler*, time-efficient garbage collection algorithms. The performance of the algorithm is further enhanced by the use of copying algorithms whenever made possible by the deterministic nature of the executing program.

1 Introduction

This report presents the design of the Aquarius Prolog garbage collector. The fundamental goal of the Aquarius project is to establish the principles by which very large improvements in performance can be achieved in machines specialized for calculating difficult problems in design automation, expert systems, and signal processing. It is currently focusing on an experimental multiprocessor architecture for the high performance execution of Prolog.

Our goal was to design a fast garbage collector for Aquarius which does not have the following problems associated with existing algorithms for Prolog: poor locality and thus poor virtual memory performance, excessive complexity of the design, reduction in addressing capability due to the need for garbage collection bits.

To achieve high performance and good locality, we adapted to Prolog the principles of generation based garbage collection which were developed for Lisp and Smalltalk [LH83,Moo84,Ung87,Sha87].

These techniques are based on the fact that most garbage cells are to be found among newly created objects. A garbage collector that concentrates its efforts on newly allocated objects can have high locality, low cpu requirements, while recovering most of the unused space.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
15 AUG 1988		**00-00-1988 to 00-00-1988**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Prolog Garbage Collector for Aquarius	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report presents the design and evaluation of the garbage collector we designed for the Aquarius project. Our design is the result of an attempt to incorporate into Prolog implementations the ideas which made generation scavenging successful for Lisp and Smalltalk. The main challenge was to take advantage of generation scavenging without giving away the basic Prolog technique of memory recovery upon backtracking based on stack deallocation. We were able to do so with little extra overhead at run-time. Our main strategy consists in restricting the action of the garbage collector to a fixed amount of memory allocated at the top of the global stack. This strategy has several advantages: it improves the locality of the executing program by keeping the data structures compacted and by allocating new objects in a fixed part of the address space; it improves the locality and the predictability of the garbage collection, which can concentrate its efforts on the fixed size area where new objects are allocated; and it allows us to use simpler, time-efficient garbage collection algorithms. The performance of the algorithm is further enhanced by the use of copying algorithms whenever made possible by the deterministic nature of the executing program.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**54**	

Our garbage collector is based on several of these techniques. In our scheme, as in *generation scavenging* [Ung87], new objects are allocated on an area of a fixed size, in a fixed memory location, and the garbage collector is called when this area is filled up with new objects. We support only two generations, *old* and *new*, and we only garbage collect the objects once. We argue, in section 2.2 that this simple strategy is the most suitable in the case of Prolog. Our algorithm is described in more detail in section 3

Another factor which contributes to the speed of advanced Lisp and Smalltalk garbage collectors is the use of algorithms based on copying, as opposed to the slower marking and compacting technique. The main drawback of copying algorithms is that they do not conserve the order of creation of objects, while marking and compacting algorithms do. This makes the use of copying algorithms difficult in Prolog, since global objects are usually created on a global stack, to allow fast memory recovery when backtracking.

Using heap allocation of data structures in Prolog would allow the use of fast copying algorithms for garbage collection. Unfortunately, it would make the recovery of storage on backtracking much slower, and the implementation not significantly simpler. To get the best of both worlds, we suggest instead to keep the stack allocation of global objects and to make use of copying algorithms to garbage collect deterministic computations. We describe in section 4 how we were able to obtain significant speedups that way.

In section 5, we discuss what kind of hardware and operating support could be used to speed up our algorithms. In section 6, we introduce some other issues relevant to Prolog garbage collection: among them, a an assessment of virtual backtracking. Finally, in section 7, we review previous work on Prolog garbage collection and compare it with our scheme.

2 Prolog Specifics

In the next subsection, we give a short introduction to the Warren Abstract Machine (WAM) [War83]. This subsection can be skipped by the reader if he or she is familiar with the WAM. In the following subsections, we discuss in more detail those features of the WAM which have a strong influence of the design of the garbage collector.

2.1 Basic Notions on the WAM

2.1.1 Memory Layout

The WAM subdivides the memory into four areas: the *environment* stack, the *choice point* stack, the *global* stack and the *trail* stack. The environment stack and the choice point stack may or may not be interleaved, depending on the exact details of the implementation.

Environments and choice points are described below. The global stack contains all the objects created during the execution of the program which either survive the procedure instantiation which created them, or are of a complex type (list or structure). It plays a role similar to the heap in Lisp implementations.

The trail stack contains the addresses of the variables that have to be reset when backtracking occurs. It is similar to a database log file.

2.1.2 Environments

Environments are the WAM equivalent of activation records in procedural languages. They contain the necessary information to implement procedure calls. They do not contain procedure arguments, as arguments are passed in registers. They contain only those local variables which have to survive a procedure call.

2.1.3 Choice Points

Choice points are records which save enough information about the a previous state of computation to allow backtracking to come back to this state. They contain the values of all the stack pointers at a given state of computation, as well as the contents of the active data registers. To backtrack to a given state, the WAM resets all the variables whose addresses have been logged into the trail stack since that state, then pops the stacks and restores the data registers to their previous values.

2.1.4 Variables and Untyped Pointers

Prolog variables can be seen as a special kind of data type. A Prolog variable can be in one of two states: *bound* or *unbound*. The two operations which can be performed on a variable are to bind it to a value or to read its value. A variable can be bound to any data object, including another variable, except that binding a variable to itself has no effect. Once a variable is bound, it cannot be bound again, unless it is reset to unbound by backtracking. The value of a variable is some unique identifier when it is unbound, or the value of the object it is bound to.

In the WAM, each variable is associated with a memory location (or a register for temporary variables). If the variable is unbound, it is usually implemented with a self-referencing pointer. If the variable is bound, it contains either the object or a pointer to the object it has been bound to. In case the object is itself an unbound variable, the variable is set to contain an *untyped* pointer pointing to this variable. This mechanism may create long chains of *untyped* pointers to be dereferenced at each access, but rarely does in practice [Tic85].

2.2 Automatic Memory Deallocation and the Need for Garbage Collection

One characteristic of Prolog programs, as opposed to Lisp or Smalltalk programs, is that upon query completion, the space used by the data structures created during execution of the query is automatically recovered, without the need for garbage collection. Moreover, within a query, many programs deallocate by backtracking a large fraction of the memory cells they allocate, as illustrated in table 1.

The programs we used in this simple experiment are as follows: BOYER, and BROWSE are Prolog versions of the corresponding Lisp Gabriel benchmarks [Gab85]; CHAT is a natural language parser,

PROGRAMS	*Total Allocated*	*Maximum Used*	*Percentage*
boyer	2.331	2.331	100.0%
browse	0.896	0.065	7.2%
chat	0.731	0.006	0.8%
compiler	6.648	3.467	52.2%
quicksort	24.161	12.651	52.4%

Table 1: Memory Usage in MB

parsing 16 sentences; COMPILER is a version of the Berkeley Prolog compiler compiling a program of 225 clauses and 87 procedures; quicksort is the quick sort program sorting a list of 23880 small integers.

Those programs which solve a sequence of queries (CHAT) or rely on backtracking to explore a search space (BROWSE) do not require garbage collection, while more deterministic programs (COMPILER, BOYER, QUICKSORT) do. The memory recovery observed in COMPILER and QUICKSORT is mainly due to shallow backtracking. Better compilation technology than the one we currently use will increase those percentages, by avoiding to allocate extra storage on the global stack before a clause has been selected in a deterministic procedure call.

Though in practice large programs display behaviors of arbitrary complexity, we expect that deterministic programs, or deterministic parts of large programs, are more likely to be in need for garbage collection, since they do not have other means of recovering heap space. In section 4, we give techniques to speed up garbage collection for deterministic programs.

2.3 The Trail Stack Contains Pointers from Old to New Objects

To speed up garbage collection, it is important to reduce the amount of memory that has to be scanned to find all the references into the memory area to be garbage collected [Ung87,Sha87].

Previous researchers [AHS87,BM86] have pointed out that it is possible to exploit the backtracking mechanism to reduce the amount of memory that has to be scanned on garbage collection. The main observation is that if one wants to collect the part global stack allocated since the creation of a choice point C, it is only necessary to scan the portion of the stacks allocated since the creation of C. The backtracking mechanism assures that all pointers from locations allocated before the creation of C pointing to locations created after the creation of C are accessible indirectly through entries in the trail stack allocated after the creation of C.

The main drawback with this approach is that we cannot predict in general the amount of memory above the topmost choice point. In the extreme case of a deterministic program, the use of choice points is trivial (shallow backtracking), and garbage collection cannot be localized. Moreover, as we pointed out previously, deterministic programs are those programs which are most likely to need garbage collection.

To be able to use local garbage collection in a more general situation, we still have to guarantee that pointers from an older location to a newer location are recorded in an appropriate log. This log is to be processed during garbage collection to limit the amount of memory to be scanned. A similar technique is used in [Ung87].

In a Prolog implementation, the obvious choice is to use the trailing mechanism to that purpose. We decided to record in the trail every variable binding, which guarantees that the trail stack contains the addresses of all pointers from old objects to new objects, and to leave to the garbage collector the role of garbage collecting the trail stack. We discuss this issue further in section 6.6.

2.4 Living, Preserved and Dead Objects

Garbage collectors for Lisp or Smalltalk have only to deal with two kinds of objects: the *living objects*, which are reachable from the registers, the control frames, and the global variables, and the *dead objects*, which are not.

Backtracking introduces a third kind of object which we call *preserved objects*. A *preserved object* is an object that is not directly accessible from registers and control frames, but is accessible from the register values saved in choice points, or from the inactive parts of control frames saved by the backtracking mechanism.

Let $\mathcal{S}_0$ be the initial state of the program, $\mathcal{S}_n$ the present state of the program, and $\mathcal{S}_1, \ldots, \mathcal{S}_{n-1}$ be the intermediate states of the program currently saved by the backtracking mechanism. For any preserved object, there is some integer $k < n$, such that the object is accessible from state $\mathcal{S}_k$ but is not accessible from state $\mathcal{S}_{k+1}, \ldots, \mathcal{S}_n$. Any modification to the preserved object that may have occurred since state $\mathcal{S}_k$ will be undone by backtracking before the object is ever referenced by the program again. In particular, pointers contained in the preserved object that are newer than $\mathcal{S}_k$ need not be followed by the garbage collector. Instead, these pointers can be reset to unbound. This operation, akin to backtracking, is known as *virtual backtracking* [PB85,AHS87].

Virtual backtracking is an optimization of the garbage collector and may be ignored. In that case, the garbage collector does not distinguish between preserved objects and living objects. In section 6.5 we present a simple implementation of virtual backtracking derived from [AHS87] which does not require marking bits, and we discuss the importance of this optimization.

2.5 The Logical Variable

The WAM implementation of Prolog logical variables makes use of untyped pointers to data structures, which are normally automatically dereferenced when encountered, except when they are self-referencing. Self-referencing pointers are used to implement unbound variables. These pointers are guaranteed to always point from new to old objects, to prevent dangling references from occurring on backtracking.

This ordering constraint and the presence of self-referencing pointers complicates the algorithms based on copying or on pointer reversal techniques (see for example, the adaptation to Prolog of

Morris' algorithm [Mor79] in [AHS87]). Self-referencing pointers make variable sharing easy, but relocation more difficult.

This ordering constraint does not need to be respected when the variable and the object it points to are not separated by a choice point. A copying garbage collector can take advantage of this fact.

2.6 Initialization of Local Variables

Local variables have to be initialized before each call to the garbage collector, to avoid the presence of dangling references when the garbage collector is invoked.

In current WAM implementations, programs can only create a constant amount of objects between two procedure calls. It is therefore sufficient to check for stack overflow at procedure entry. This guarantees that the garbage collector will only be called at procedure entry. Consequently, we only have to guarantee that variables are correctly initialized at procedure entry, which allows us to dramatically reduce the number of initialization instructions that would be required otherwise, as illustrated in table 2. An example of the optimization performed by the compiler is given below.

```
allocate 2                  ⟹   allocate                  ⟹   allocate
get_variable Y1,X1              init Y1                       init Y2
call foo                        init Y2                       get_variable Y1,X1
put_value Y1,X2                 get_variable Y1,X1            call foo
put_variable Y2,X3              call foo                      put_value Y1,X2
                                put_value Y1,X2               put_value Y2,X3
                                put_variable Y2,X3
```

It should be noted that, when the first use of a local variable occurs after a procedure call, the use annotation should be a *value* annotation instead of a *variable* annotation to ensure that the binding is trailed, as illustrated in the preceding example. This is necessary in the case the procedure call leaves a choice point, and the local variable is bound to an object created on the global stack above this choice point. In this situation, the variable contains a pointer from an object created before a choice point pointing to an object created after the same choice point, and the garbage collector relies on the fact that every such pointer has been trailed to avoid scanning the bottom part of the environment stack.

Some built-ins require an unbounded amount of space on the global stack (e.g. *retract*). Since they should also check for overflow, and may invoke the garbage collector, their invocation should be considered as a procedure call by the optimizer.

3 A Simple Garbage Collector

In this section, we will first present a simple version of our garbage collector. This algorithm is easy to implement, does not require the complexity of algorithms based on pointer reversal techniques, and displays good performance and locality.

PROGRAMS	*Before*	*After*	*Removed*
boyer	19	0	100.0 %
browse	34	7	79.4 %
chat	768	85	88.9 %
comp	977	105	89.3 %
qsort	11	2	81.8 %

Table 2: Initialization Optimization

We will first introduce some terminology that we will use throughout the rest of this paper. Then we will describe our algorithm in more detail, and finally present some performance results.

3.1 Terminology and Basic Concepts

Our basic terminology is similar to the one used by [Sha87].

- `new space` is the part of the global stack, of fixed size, in which new objects are allocated when created. The garbage collector only garbage collects `new space`. After each garbage collection, the remaining living cells are added to `old space`. In our scheme, we lock `new space` at the top of the address space allocated to the global stack. This has the double advantage of increasing the locality of the executing program and simplifying the garbage collector; a similar trick was used by Ungar [Ung87]. H is the pointer pointing at the next free location of `new space`.

- `old space` is the part of the global stack that contains all the data objects that have survived a garbage collection. It is formed of one contiguous segment of address space, at the bottom of the global stack. Its size is only limited by the size of the address space. H2 is the pointer pointing at the next free location of `old space`.

- `copy space` is the part of the global stack just above `old space` in which the garbage collector copies the surviving objects. It is added to `old space` when the garbage collector completes its work.

- `base space` is the part of the memory which contains references to objects in `new space` that have to survive a garbage collection. In the case `new space` starts at a choice point boundary, `base space` is only composed of the part of the environment stack, the choice point stack above this choice point, as well as the memory locations pointed to by entries in the trail stack above this choice point [BM86,AHS87]. Since in general `new space` does not start at a choice point boundary, we keep track of stack pointer variations to determine the exact limit of `base space`. This is explained in more detail in section 3.3.

We choose the simplest possible design by making objects in `new space` which survive only one garbage collection elements of `old space`, and therefore no longer candidates for subsequent

NEW SPACE	BOYER		COMPILER	
IN KB	*global*	*trail*	*global*	*trail*
16	24.8 %	0.4 %	17.1 %	3.4 %
32	22.4 %	0.3 %	11.2 %	1.9 %
64	20.4 %	0.1 %	9.0 %	1.5 %
128	18.6 %	0.1 %	6.8 %	1.0 %
256	16.8 %	0.0 %	5.4 %	0.8 %
512	13.8 %	0.0 %	4.0 %	0.6 %
1024	11.8 %	0.0 %	2.6 %	0.6 %
2048	11.5 %	0.0 %	2.4 %	0.6 %

Table 3: Garbage Collection Survival Rate

garbage collections. In other words, we only support two generations of objects: *old* and *new*. We believe that this is the adequate choice for Prolog for two reasons. First, most objects do not survive their first garbage collection, as illustrated in table 3. (Our results agree with similar studies for Lisp and Smalltalk. For Lisp, Shaw found the proportion of new objects which survive their first garbage collection to be between 10% and 30% with a **new space** size of 32KBytes. For Smalltalk, Ungar found 20% of survivors after their first garbage collection with a **new space** size of 20KBytes [Ung87]). By garbage collecting objects only once, we can expect to recover 70 to 90% of the garbage cells. Second, Prolog programs have other means of recovering memory, through backtracking or query completion. We thus do not have to worry as much about the management of very old objects.

An important point to note is the very low survival rate of pointers in the trail stack. The trail stack should not be neglected; the amount of trail stack scanned by the garbage collector is roughly 85% of the amount of global stack scanned for BOYER, and 45% for COMPILER. While interpreting these data, it should be kept in mind that our implementation trails every variable binding.

3.2 Invocation Mechanism and Area Overflow

The garbage collector can only be invoked at procedure entry or inside the few built-ins which are not guaranteed to allocate a fixed amount of memory. This has the advantage of reducing the number of stack overflow checks, when not detected by hardware, and allows us to optimize the local variable initialization code, as mentioned above.

At procedure entry, H is checked against the address space limit. On overflow, the garbage collector is called. To prevent overflow from happening between two procedure calls, a provision for overflow is made at the top of **new space**.

Builtins which may create large objects should either know in advance how much memory they will need, or be restartable. The case of a built-in needing more space than an entire **new space** should be handled properly. The simplest solution is to first garbage collect **new space**, and then

allocate the new object on the top of old space.

3.3 Bookkeeping and Overhead on Normal Execution

Our scheme requires the use of several additional registers to maintain information on which part of the memory needs to be scanned at the next garbage collection. These registers need not be saved in choice points.

The bookkeeping required by our algorithm is as follows:

1. we need to maintain two heap pointers, H and H2, instead of one. H2 needs only be updated in the case when failure deallocates new space entirely which is a rare event. (with new space of size 16KB, this event only occurred 109 times in the COMPILER benchmark). Just give the data for comp and boyer, and for window = 32)

2. there is a similar bookkeeping to perform for the two trail stack pointers TR and TR2. TR2 points at the top of the stack, while TR2 contains the lowest value of TR since the last garbage collection.

3. we also maintain a E and a E2 pointer. The E points to the current environment, which may not be at the top of the stack. The E2 pointer contains the lowest value of E since the last garbage collection. There is more overhead associated with the E pointer, since it has has also to be maintained on environment deallocation (the deallocate instruction in the WAM).

The use of E2 is not as crucial than the use of H2 and TR2, since the environment stack is typically much smaller than the global stack or the trail stack. It could be dispensed of, at the cost of some unnecessary scans of environments during garbage collection.

There is no need to maintain B2 pointers for choice points, since this information can be easily retrieved from the H2 or E2 pointers by the garbage collector.

3.4 Marking

Our marking algorithm is straightforward. It does not implement virtual backtracking, and treats preserved objects as dead objects. We will discuss in section 6.5 how it is possible to extend our algorithm to implement virtual backtracking.

Marking proceeds recursively from all the pointers in base space pointing into new space. It does not need to visit the objects outside new space it may encounter [BM86,AHS87]. The locality of base space and new space guarantees the locality of our marking algorithm. During this phase, we can use copy space as a recursion stack for recursive marking, which is simpler and faster than its more space efficient alternatives [Coh81]. We discuss the use of marking bits in section 6.4.

WINDOW SIZE	*elapsed time*		*page faults*		*speedup*
(KB)	*average*	*90 percentile*	*average*	*90 percentile*	
16	185.4	0.74 %	0.00	± 0.00	1.31
32	184.1	0.52 %	0.00	± 0.00	1.32
64	182.7	0.33 %	0.00	± 0.00	1.33
128	182.7	0.74 %	0.00	± 0.00	1.33
256	181.3	0.21 %	0.00	± 0.00	1.34
512	184.0	1.11 %	4.14	± 5.98	1.32
1024	215.9	1.09 %	360.43	± 29.46	1.13
2048	243.1	2.33 %	752.00	± 31.22	1.00

Table 4: Paging Performance with the Boyer Benchmark

3.5 Compacting

The compacting phase of our algorithm simply slides down the marked cells in **new space** to copy space, updates the internal pointers on the fly, and leaves behind in **new space** the corresponding relocation addresses. Unmarked cells are also overwritten with the relocation address of the most recent marked cell encountered; this is for simplifying the next phase of the algorithm.

3.6 Updating

The algorithm finally rescans **base space** in search for pointers to **new space** to be updated to their new value. It uses the relocation table now contained in **new space** for that purpose. It also updates the global stack pointer (H) choice point entries by using the relocation table.

3.7 Performance Results

We measured the paging and elapsed time performance of our algorithm on a Sun 3-50, with 3.2MB of physical memory, running Sun Unix 4.2 release 3.2. The benchmark used is BOYER. We varied the size of **new space** from 16 to 2048 KBytes. The benchmark was run 7 times; we give the average results as well as the confidence interval for the 90 percentile. The results are given in table 4.

We believe that these results are not as dramatic as they should be for a faster system. Our measurements were taken on a byte-code emulator, which performance is roughly 3 to 6 times slower than Quintus Prolog. All other things being equal, speeding up our emulator by a factor of 3 in program execution would make the lack of locality of the exhaustive garbage collector look worse than the more local garbage collectors by a factor of 2 instead of 1.3.

For high-end machines, the lack of locality is even more costly. In a previous implementation of our algorithm, also on a relatively slow byte-code emulator, we observed a reduction by a factor of 20 of the number of page faults on an IBM 3081, and a speedup of the program by a factor of

1.5 for a maximum new space size of 512KB. Unfortunately paging measurements for time sharing systems are heavily dependent on the load, and thus are not very accurate.

4 Taking Advantage of Copying Algorithms

The algorithm we introduced in the previous section displays much higher locality than exhaustive garbage collectors. By reducing paging, it is has the potential of reducing the *elapsed time* of spent in program execution. To increase the performance of the garbage collector in terms of cpu time, we propose to take advantage of copying algorithms.

We first propose a simple algorithm which takes advantage of copying only when the entire new space is above the topmost choice point. We then investigate a more general way to incorporate copying into our algorithm to extend its scope of applicability.

4.1 A Simple Scheme using Copying Algorithm

This simple scheme works as follows: whenever the garbage collector is called, it tests to see whether the entire new space is above the topmost choice point. If it is the case, it uses a copying algorithm to perform garbage collection. It is possible to do so since in that case the relative order of the data structures in new space need not be maintained. Otherwise, it uses the marking and compacting algorithm we presented previously. We describe the copying algorithm we use in more detail in the next subsection.

4.1.1 The Copying Algorithm

The copying algorithm we used is directly derived from classic copying techniques ([Che70,Bak78]). It proceeds as follows: for each pointer into new space pointing to an unmarked object, the object pointed to is copied into old space. The original copy is marked and replaced by relocation pointers pointing to the corresponding locations into old space. Pointers to marked locations are immediately relocated.

The only potential difficulty in adapting copying algorithms to Prolog is due to the presence of untyped pointers. An untyped pointer can point to an element of a structure or a list. In a straightforward implementation, if the copying algorithm encounters first the pointer and later the structure or the list, the cell pointed to by the untyped pointer will be copied twice. In the present case, however, the entire new space is guaranteed to be above the topmost choice point. Therefore there is no need to maintain untyped pointers into new space to guarantee the correctness of the backtracking mechanism. Our algorithm simply removes all untyped pointers pointing into new space.

WINDOW SIZE	*mark & compact*		*copy*		*speedup*
(KBYTES)	*average*	*90 percentile*	*average*	*90 percentile*	
16	4.51	0.55%	3.37	1.07%	1.34
32	4.17	1.08%	3.04	1.61%	1.37
64	3.95	0.84%	2.87	1.06%	1.38
128	3.62	1.17%	2.59	0.42%	1.39
256	3.27	1.50%	2.31	1.17%	1.41
512	3.04	1.30%	2.10	1.67%	1.45
1024	2.95	1.01%	1.97	1.61%	1.50
2048	2.99	1.20%	2.03	1.07%	1.47

Table 5: Performance of Improved GC with the Boyer Benchmark

4.1.2 Performance

We compared the efficiency of the copying algorithm with our previous algorithm based on mark and compact. We used BOYER as a benchmark and the Unix *getrusage* system call to measure the time spent in the garbage collector. The benchmark was run 6 times. We give the average results as well as the confidence intervals for the 90 percentile in table 5. In this measurement, the program was entirely deterministic, and only the faster copy algorithm was used by the enhanced algorithm.

By comparing the data of table 5 with the data of table 3, we can see that the experiments confirm the fact that copying perform better with lower survival rates. For example, similar experiments with the QUICKSORT benchmark yields speedups of up to 1.86 for a survival rate of 4.9%, with 1024KB allocated to **new space**.

4.2 An Improvement on the Simple Scheme using Copying Algorithm

We can extend the scope of applicability of the copying algorithm as follows. At each garbage collection call, we interweave marking and copying. Copying is used whenever a pointer to the first choice point segment in **new area** is encountered; otherwise marking is performed. There is little difficulty in doing so since marking and copying can follow the same order of traversal of the program data structures. We will give more details on this technique in the next section. We present our performance results in the following section.

4.2.1 The Extended Copying Algorithm

Once marking has completed for the upper part of **new space**, copying has completed for the lower part. Compaction can be then be performed for the upper part.

It is straightforward to interweave marking and copying, since both algorithms proceed the same way, by traversing recursively a data structure. Our implementation uses both a stack for

WINDOW SIZE	*mark & compact*		*mark & copy*			*speedup*
(KBYTES)	*average*	*90 percentile*	*copy mode*	*average*	*90 percentile*	
16	7.68	1.01%	42.2%	7.30	1.41%	1.05
32	5.36	0.91%	34.7%	4.97	1.82%	1.08
64	4.57	1.09%	23.3%	4.38	1.73%	1.04
128	4.06	1.20%	20.9%	3.80	1.45%	1.07
256	3.63	1.00%	12.7%	3.56	1.66%	1.02
512	3.13	0.70%	14.3%	2.96	1.81%	1.06
1024	2.97	0.75%	1.6%	2.92	2.96%	1.01
2048	1.99	0.99%	0.0%	2.04	2.77%	0.97

Table 6: Mark and Copy over Mark and Compact: Speedup

recursive marking, and Cheney's algorithm queue for copying. Processing a reference to **new space** is completed when both the stack and the queue are empty.

It is no longer possible in general to skip untyped pointers, as in the case the entire **new space** is above the topmost choice point. To avoid copying a structure or a list cell twice, we need now to delay the processing of untyped pointers pointing into the lower part of **new space**. Only the copying of the cell referenced by an untyped pointer is delayed. If this cell contain a typed pointer to a Prolog object, this object is copied and the pointer is relocated without delaying.

With virtual backtracking, it is necessary to ensure that cells which copied is delayed are marked when first visited. The reason is that virtual backtracking relies on the fact that the garbage collector marks every object accessible from states $\mathcal{S}_k, \ldots, \mathcal{S}_n$ before resetting all bindings made after $\mathcal{S}_k$ to unmarked objects, and breaks down if marking of some cells is delayed.

4.2.2 Performance

The algorithm described previously has more overhead than the simple, fast copying algorithm we introduced in the previous section. The main performance degradation comes from the overhead of deciding whether a pointer to **new space** points into the lower part or the upper part. We estimate the performance degradation to be of the order of 10%.

The other main factor which determines the overall performance of our enhanced algorithm is the percentage of cells which are collected with the copying algorithm. This percentage needs to be relatively high for us to be able to obtain a significant speedup (this is an instance of application of Amdhal's law [Amd67]). Unfortunately, this percentage decreases with the window size. For large window sizes, mark and compact may indeed be faster than mark and copy due to the extra overhead inherent to the mark and copy approach.

The experimental data confirms this analysis, as displayed in table 6. Our data have been obtained with the COMPILER benchmark described in section 2.2.

(The poor performance of the mark and copy algorithm on the COMPILER benchmark, which

PROGRAMS	*Quintus run time*	*GC cpu time*	*ratio*
boyer	16.1	2.3	14.2 %
compiler	82.6	3.6	4.4 %
quicksort	85.5	8.6	10.1 %

Table 7: GC Cputime Overhead

is an essentially deterministic program, is surprising. Closer look at the program indicated that in many parts of the program, choice points were not removed as early as possible.)

4.3 Overall Performance

To corroborate our claims of efficiency, we measured the cpu overhead of our garbage collection algorithms, for a size of **new space** of 256KB, as compared to the run time of Quintus Prolog of the same benchmark. The measurements where taken on a VAX 8600, running Quintus Prolog 1.6. They are given in table 7. The measurements are only indicative, since the two implementations may not use the same data structure representations. Quintus Prolog was given enough space to execute without garbage collection.

Given the survival rates in table 3, we can interpret these data as follows: for the memory intensive BOYER benchmark, the garbage collector increases the size of the global stack by a factor of 6.0 for a cpu overhead of 14.2%. For the COMPILER benchmark, it extends the global stack by a factor of 18.5 for a cpu overhead of 4.4%.

5 Hardware and Operating System Support

Our algorithm is simple, and does not need any specialized hardware support to run with good performance. However, there are some simple primitives that would speed it up if they were made available to the program.

The first and most obvious remark is that the hardware should provide enough general purpose registers for the software to be able to permanently allocate the H2, E2 and TR2 pointers in registers without forcing the compiler to generate too many spills to memory. We believe that 32 registers should be enough for that purpose.

The second hardware feature that will enhance performance is to have stack overflow checks done in hardware. This could be done on page boundaries only without affecting performance. In other words, this could be implemented by providing basic hardware support for virtual memory and putting the responsibility of detecting stack overflow on the operating system.

Basic hardware support for virtual memory could also be used by the garbage collector to make unnecessary the use of the E2 pointer, and to limit trailing to those bindings which need to be undone on backtracking. The operating system could easily be modified to indicate on request to

the garbage collector which pages have been modified since the last garbage collection, as suggested by Shaw [Sha87]. Only these pages may contain pointers to **new space**, and thus only these pages need to be scanned.

6 Side Issues

6.1 Interaction with Virtual Memory

The interaction between Prolog systems and virtual memory is not as friendly as one might expect. One problem originates from the stack allocation mechanism. After a deep failure, or after query completion, a large chunk of the global stack is deallocated. Unfortunately, there is usually no way to tell the virtual memory system that the corresponding pages of a deallocated chunk of virtual address space can be freed. As a consequence, unnecessary page faults will occur when the next query is started. This phenomenon has been studied by Ross and Ramamohanarao [RR86]. Our garbage collector, by constantly compacting the data structures of the executing program reduces this effect.

6.2 Applying Copying to Several Choice Point Segments

One possible generalization of our mark and copy algorithm is to apply copying to more choice point segments that just the last one. There are two difficulties with this scheme: the first one is that we cannot determine the final address of an object living in a choice point segment without having completed the garbage collection of all the segments under it. Copying can still be used, but a second pass is then required on the survivors as well as on **base space** to relocate the objects to their final position, making copying less attractive.

The second difficulty is that the algorithm needs to know into which choice point segment a given pointer points. This induces an extra cost of the order of $(1 + \log cp)$, where cp is the number of choice point segments garbage collected by copying. For these two reasons, we do not think that this approach can lead to any significant speedup over our basic mark and compact algorithm.

6.3 Choosing the Size of New Space

Using too large a **new space** will cause poor locality, as illustrated in table 4. On the other hand, using too small a **new space** will increase the survival rate and cpu time consumed by the garbage collector as illustrated in table 3 and table 7. The right compromise is system dependent. Some of our previous experiments indicated as adequate a value of 32KB for mainframe computers, 256KB to 512KB for workstations. There is no difficulty in letting the programmer adjust the size of **new space** to his or her specific needs.

6.4 Marking Bits vs. Marking Table

It is entirely possible to implement our algorithm using marking bits. Only one bit per word is necessary. In fact, our first implementation was designed that way. In the present implementation, we experimented with the use of a marking table. The use of a marking table is made possible by the fact that the size of **new space** is fixed. Since our implementation was targeted towards 32-bit word, byte-addressable general-purpose machines, we decided to use one byte of mark per word in **new space**. This causes a space overhead of $\frac{1}{4}$ of the size **new space**, but allows faster access to the marks.

The overhead of initializing the mark table can be asymptotically reduced by a factor of 255 by using a rotating mark, incremented at each consecutive call to the garbage collector, in the sequence $(1, \ldots, 255, 1)$.

6.5 Virtual Backtracking

We implemented virtual backtracking in all of our algorithms. We basically used the scheme described in [AHS87]. The main difference is that our scheme does not mark the environments to keep track of previous visits during the marking phase.

It is possible to maintain one extra environment pointer to determine which part of a given environment has already been visited. As the environment chain protected by a given choice point is visited, this pointer is set to point to the top of the chain of the previous choice point. It is moved down the environment stack to the last environment in its chain before the environment where the two chains merge. We suppose the Prolog system allows us to determine the size of an environment as viewed by any of its children by looking at some fixed offset of the return address saved in the child environment. This condition is fulfilled in the WAM.

Our experiments with virtual backtracking were disappointing, as we did not find any advantage in using it for most of our benchmarks. Only with CHAT and a **new space** size of 4KB we were able to obtain some improvement, with a very high survival rate of 80.5% was somewhat reduced to 75.1% by virtual backtracking; with a size of 8KB, the garbage collector was not even called. Our conclusion is that we did not find sufficient evidence that the extra complexity of virtual backtracking is worth implementing, with the usual caveat that we only check on a few large programs.

6.6 Trailing every Binding

Trailing every binding does not necessarily cause overhead. Whether it does depends on two factors: the relative cost of doing the trailing and of checking whether it is necessary to trail; the proportion of trails which can be avoided at *binding time*. On many machines, checking will be more expensive than trailing.

In [TD87], Touati and Despain made some measurements on the proportion of trails which can be avoided at binding time. The proportion varied widely from benchmark to benchmark, but were

found between 30% to 70% for most benchmarks. The surviving trail entries will have to be checked a second time during garbage collection. It may be more efficient to delay the check on trail entries by trailing every binding, and recovering the storage on garbage collection. As the data in table 3 indicate, at garbage collection time we can predict the output of the check with high accuracy.

7 Comparison with Previous Work

The first attempt in using the ideas from generation scavenging was [PB85]. Their approach was based on choice point segments. For each choice point, they allocated a different logical segment of memory, and used this choice point segment as a unit to perform garbage collection. We believe that this scheme is more complex than ours, and more disruptive of the basic WAM organization. Unfortunately, the authors do not give performance data.

The first study to notice that garbage collecting above the topmost choice point is significantly simpler than the general case was [BM86]. The main difference between theirs and ours is the trap mechanism: ours is automatic, determined dynamically by window overflow; theirs requires the intervention of the programmer.

Several Prolog implementations [BBCT86,NI86,AHS87] have used the pointer reversal techniques introduced by [Mor78]. The most recent WAM garbage collector we are aware of was described in [AHS87]. Though the authors are aware of the fact that partial garbage collection performs much better than exhaustive garbage collection, they do not propose any scheme to exploit it. Moreover, their design is still based on compaction algorithms that do not use extra memory space, which are more complex, and, we believe, slower than ours.

8 Conclusion and Future Work

We designed and implemented a Prolog garbage collector than displays good locality and high cpu performance. We modified our Prolog implementation to allocate new objects at the top of the global stack address space, in an area of fixed size. By calling the garbage collector each time this area overflows, we were able to ensure good locality. By making use of copying algorithms rather than marking and compacting algorithms on appropriate parts of the area to be garbage collected, we were able to significantly improve the cpu performance of our algorithm on deterministic programs. We were disappointed with our experience with virtual backtracking, where the extra implementation complexity does not seem to pay off.

Future work will include the assessment of multiple generation schemes, as well as the study of the relationship between garbage collection and the dynamic resizing of stacks as originally done in DEC-10 Prolog [WP77].

Acknowledgements

We would like to thank Yasuo Asakawa, David Bowen, Chien Chen, Bruce Holmer, Hideaki Komatsu, Tim Lindholm, Richard O'Keefe, Naoyuki Tamura, Peter Van Roy, Jim Wilson and Ben Zorn for the time they spent discussing many of these ideas and their helpful comments on earlier drafts.

Partial support for this work was generously provided by the California MICRO program, the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871, Monitored by Space and Naval Systems Warfare Command under Contract No. N00039-84-C-0089, and by assistance from IBM Corporation, Digital Equipment Corporation and NCR Corporation. Part of this work was performed when the first author was visiting IBM Tokyo Research Laboratory during the summers of 1986 and 1987 and published in [TH88].

References

[AHS87] K. Appleby, S. Haridi, and D. Sahlin. Garbage collection for prolog based on wam. Tr, IBM, IBM Thomas J. Watson Research Center PO Box 218 Yorktown Heights, NY 10598, June 1987.

[Amd67] G.M. Amdhal. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[BBCT86] K. A. Bowen, K. A. Buettner, I. Cicekli, and A. K. Turk. The design and implementation of a high-speed incrementable Prolog compiler. In E. Shapiro, editor, *Third International Conference on Logic Programming*. Springer Verlag, Lecture Notes in CS 225, July 1986.

[BM86] J. Barklund and H. Millroth. Garbage cut for garbage collection of iterative Prolog programs. In *3th Symposium on Logic Programming*, Salt Lake City, September 1986. IEEE.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[Gab85] R. P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press series in Computer Systems. The MIT Press, 1985.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes o ,objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Moo84] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[Mor78] F. Lockwood Morris. A time and space efficient garbage collection algorithm. *Communications of the ACM*, 21(8):662–665, 1978.

[Mor79] F. L. Morris. On a comparison of garbage collection techniques. *Communications of the ACM*, 22(10):571, October 1979.

[NI86] Nishikawa and M. Ikeda. Psi no garbage collector (in japanese). TR 213, ICOT, 21F, Mita Kakusai Bldg., 4-28 Mita 1, Minato-ku, Tokyo 108, Japan, 1986.

[PB85] E. Pittomvils and M. Bruynooghe. A real time garbage collector for Prolog. In *2nd Symposium on Logic Programming*. IEEE, 1985.

[RR86] M. L. Ross and K. Ramamohanarao. Paging strategy for Prolog based on dynamic virtual memory. In *Third Symposium on Logic Programming*, pages 46–55, September 1986.

[Sha87] R. A. Shaw. Improving garbage collector performance in virtual memory. CSL-TR 87-323, Stanford University, CSL, Stanford University, Stanford, CA 94305-4055, March 1987.

[TD87] H. Touati and A. Despain. An empirical study of the Warren Abstract Machine. In *4th Symposium on Logic Programming*, San Francisco, September 1987. IEEE.

[TH88] H. Touati and T. Hama. A light-weight prolog garbage collector. In *International Conference on Fifth Generation Computer Systems 1988*, Tokyo, Japan, November 1988. ICOT.

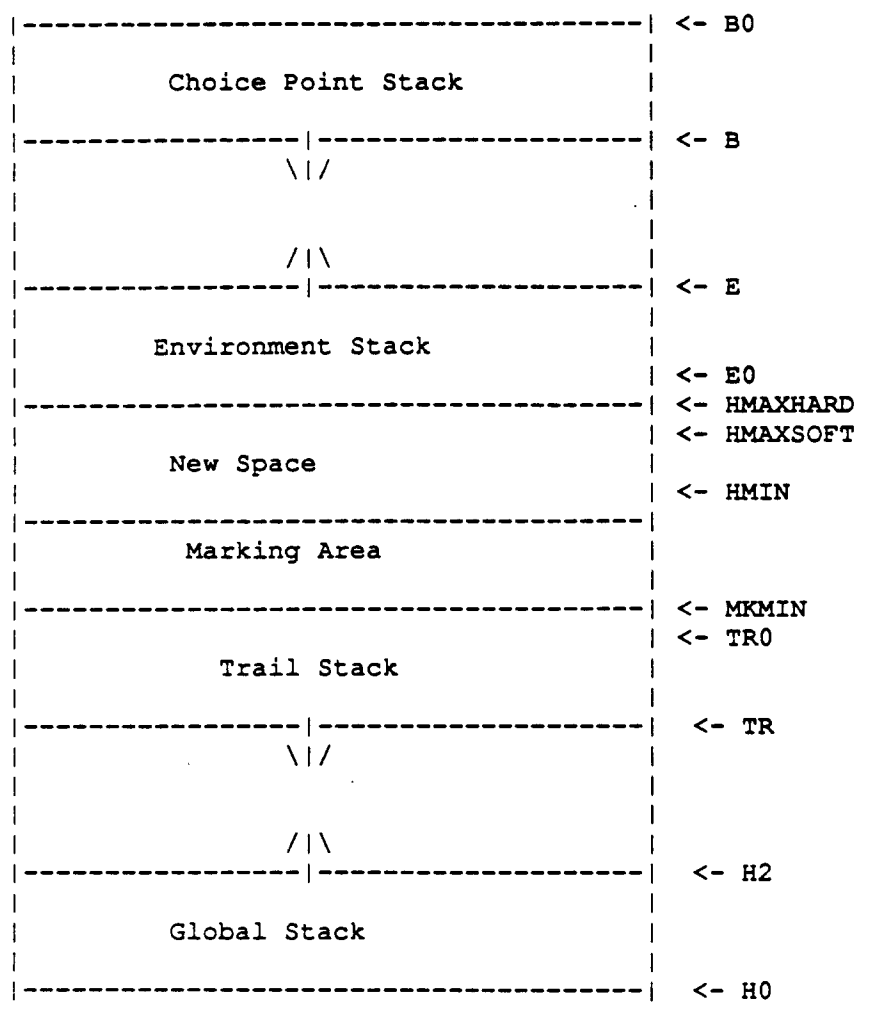
[Tic85] E. Tick. Prolog memory-referencing behavior. 85 281, Computer Systems Laboratory, Stanford University, Palo Alto, California, September 1985.

[Ung87] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* ACM Distinguished Dissertations. The MIT Press, 1987.

[War83] D. H. D. Warren. An abstract prolog instruction set. Technical report, SRI International, Arificial Intelligence Center, August 1983.

[WP77] D. H. D. Warren and L. M. Pereira. Prolog — the language and its implementation compared with lisp. In *Symposium on Artificial Intelligence and Programming Langages*, pages 109–115, August 1977.

/*



*/

```
/* Copyright Herve' Touati, 1988, Aquarius Project, UC Berkeley */

extern CellPtr S;
extern CellPtr B0;
extern CellPtr B;
extern CellPtr E0;
extern CellPtr E;
extern CellPtr TR;
extern CellPtr TR0;
extern CellPtr H0;
extern CellPtr H;
extern CellPtr R;
extern CellPtr R0;
extern InstrPtr P0;
extern InstrPtr P;

#ifdef WITH_GC
extern CellPtr H2;
extern CellPtr TR2;
extern CellPtr E2;
extern CellPtr HMIN;
extern CellPtr HMAXSOFT;
extern CellPtr HMAXHARD;
const int HMAX_SECURITY = 256;
extern unsigned char* MKMIN;
#endif

 /* points to an escape that signals successful termination */
extern InstrPtr CP0;
 /* points to a fail instruction */
extern InstrPtr FP0;
 /* points to an escape that signals unsuccessful termination */
extern InstrPtr NP0;
 /* points to the metacall escape */
extern  InstrPtr MP0;
extern  int next_instruction;
enum {
#define use(Name,ID,Coeff,Reg,Type)\
  ID,
#include "memory_sizes.h"
#undef use
  LAST_SIZE
  };
extern int memory_sizes[];
extern Cell NIL;
extern Cell LIST_FUNCTOR;

class Memory {
 public:
  void init();
  StringTable* ST;
  Memory(StringTable& table);
  void allocate();
};

#define NUMBER_OF_REGISTERS 8
extern Cell X[];

 /* layout of an environment */
 /* B  E  P  Y1  Y2  Y3 ... */
 /* -3 -2 -1  0   1   2    */
enum {
  B_ENV_OFFSET = -3,
  E_ENV_OFFSET = -2,
```

```
 P_ENV_OFFSET = -1,
 Y1_ENV_OFFSET = 0,
 };

/* position of E above the top of the stack when an env is created */
enum {
 E_TOP_OFFSET = 3
 };

/* Layout of a choice point */
/* E  H  TR  P  SIZE X1  X2  X3 ... */
/* 1  2  3   4  5    6   7   8  9     */
/* the CP stack grows donwards, so Xi are the first pushed on the stack */
/* and B points at the top of the stack */
/* A is the top of the environment stack */
/* always equals to E except for those stupid intra-clause choice */
/* points */
enum {
 E_CP_OFFSET = 1,
 H_CP_OFFSET = 2,
 TR_CP_OFFSET = 3,
 P_CP_OFFSET = 4,
 SIZE_CP_OFFSET = 5,
 X1_CP_OFFSET = 6,
 FIXED_CP_SIZE = 5
 };
```

```
/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

extern void mark_from_base(Cell*);
extern void mark_from_base_sweep(Cell*);
extern void copy_from_base(Cell*);

struct UpStack {
  Cell* sp;
  Cell* sp0;
  void init(Cell* p) {sp0 = sp = p;}
  Cell* bottom() {return sp0;}
  Cell* top() {return sp;}
  void push(Cell* val) {*sp++ = cell(val);}
  Cell* pop() {return cellp(*--sp);}
  int nonempty() {return (sp > sp0);}
};

struct DownStack {
  Cell* sp;
  Cell* sp0;
  void init(Cell* p) {sp0 = sp = p;}
  Cell* bottom() {return sp0;}
  Cell* top() {return sp;}
  void push(Cell* val) {*sp-- = cell(val);}
  Cell* pop() {return cellp(*++sp);}
  int nonempty() {return (sp < sp0);}
};

 /* basic data structure to implement Cheney's copy algorithm */
struct CopyStack {
  Cell* first;
  Cell* second;
  void init(Cell* p) {first = second = p;}
  Cell* top() {return first;}
  void push(Cell val) {*first++ = val;}
  Cell* pop() {return second++;}
  int nonempty() {return (first > second);}
};

inline Cell* max(Cell* a, Cell* b)
{
  return (a > b) ? a : b;
}

inline Cell* min(Cell* a, Cell* b)
{
  return (a < b) ? a : b;
}

extern void init_stats();
extern void display_stat1(char*, int, int);
extern void init_marking_table();
extern CellPtr B2, HMIDDLE;
extern unsigned char MARK;

 /* UTILITIES */

inline int to_new_space(Cell* p)
{ return (p < H) && (p >= HMIN); }

inline int pointer_to_new(Cell val)
{ return (get_tag(val) != TAGCONST && to_new_space(addr(val))); }

 /* better be sure p points to new space */
```

```
inline Cell* reloc_addr(Cell* p)
{ return cellp(*p); }

inline void set_reloc_addr(Cell* p, Cell* new_addr)
{ *p = cell(new_addr); }

inline Cell check_and_relocate(Cell var)
{
  int tag = get_tag(var);
  Cell* ptr = addr(var);
  if (tag != TAGCONST && to_new_space(ptr))
    return make_ptr(tag, reloc_addr(ptr));
  else
    return var;
}

 /* suppose that p is an address to a location in new space */
 /* please do the check!! note: new space contain a relocation table. */
overload relocate;
inline Cell relocate(Cell var)
{ return make_ptr(get_tag(var), reloc_addr(addr(var))); }

inline Cell relocate(int tag, Cell* p)
{ return make_ptr(tag, reloc_addr(p)); }

inline void mark(Cell* p)
{ MKMIN[p - HMIN] = MARK; }

inline int marked(Cell* p)
{ return (MKMIN[p - HMIN] == MARK); }

inline int unmarked(Cell* p)
{ return (MKMIN[p - HMIN] != MARK); }

extern void store_regs_in_env();
extern void restore_top_env();

struct Env {
  Cell* e;
  int size;
  int already_treated;
  void next() {
    size = instrp(e[P_ENV_OFFSET])->arg2; /* P points to the call instr */
    already_treated = 0;
    e = cellp(e[E_ENV_OFFSET]);
  }
  Env() {}
  Env(Cell* E) {init(E);}
  void init(Cell* E) {
    e = E;
    next();
  }
  void treated(int n) {already_treated = n;}
  void mark();
  void fast_copy();
  void mark_sweep();
  void update();
};

struct ChoiceRecord {
  Cell* tr;
  Cell* e;
  Cell* h;
};
```

```
struct Choice {
  Cell* b;
  Env already_done;
  Env preserved;
  Cell* tr;
  Choice(Cell*, Cell*);
  void next() {
    b = b + FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
    preserved.init(cellp(b[E_CP_OFFSET]));
  }
  int last() { return (b >= B2); }
  void mark();
  void mark_sweep();
  void virtual_backtrack();
  void virtual_backtrack_sweep();
  void mark_preserved_envs();
  void mark_preserved_envs_sweep();
  void update();
  void update_preserved_envs();
};

extern void cp_to_cp_forward();
extern void cp_to_cp_backward();

 /* The TRAIL STACK */

enum {
  TRAIL_SKIP,
  TRAIL_KEEP,
  TRAIL_RELOC,
  TRAIL_IND_RELOC,
  TRAIL_COPY_RELOC,
  TRAIL_MARK
  };

 /* sort of a cp cache, with some control info. */
 /* the main point is to make sure that the TR entries are updated */
 /* after the former values are read */
struct TrailCP {
  Cell* b;
  Cell* next_b;
  Cell* last_b;
  Cell* tr;
  Cell* next_tr;
  Cell* e;
  Cell* h;
  TrailCP(Cell* B2, Cell* B) {b = B2; last_b = B; init();}
  int nonempty() {return (b > last_b);}
  void init() {next_b = b;
               next_tr = cellp(b[TR_CP_OFFSET]);
               next();}
  void next() {b = next_b;
               e = cellp(b[E_CP_OFFSET]);
               h = cellp(b[H_CP_OFFSET]);
               tr = next_tr;
               next_b = b - (FIXED_CP_SIZE + b[SIZE_CP_OFFSET]);
               next_tr = cellp(next_b[TR_CP_OFFSET]);}
  void update_tr(Cell* tr) { next_b[TR_CP_OFFSET] = cell(tr); }
  int pass1_action(Cell* ptr) {
    if (ptr >= e || (ptr < E0 && ptr >= h))
      return TRAIL_SKIP;
    else if (ptr >= E2 || (ptr < E0 && ptr >= HMIN))
      return TRAIL_KEEP;
```

```
    else
      return (pointer_to_new(*ptr)) ? TRAIL_MARK : TRAIL_KEEP;
  }
  int pass2_action(Cell* ptr) {
#ifdef WITH_VIRTUAL_BACK
    if (*ptr == make_ptr(TAGREF,ptr))
      return TRAIL_SKIP;
    else if (to_new_space(ptr)) {
#else
    if (to_new_space(ptr)) {
#endif
      if (unmarked(ptr))
            return TRAIL_SKIP;
      else
            return (ptr >= HMIDDLE) ? TRAIL_RELOC : TRAIL_COPY_RELOC;
    } else
      return TRAIL_IND_RELOC;
  }
  int pass2_action_sweep(Cell* ptr) {
#ifdef WITH_VIRTUAL_BACK
    if (*ptr == make_ptr(TAGREF,ptr))
      return TRAIL_SKIP;
    else if (to_new_space(ptr)) {
#else
    if (to_new_space(ptr)) {
#endif
      if (unmarked(ptr))
            return TRAIL_SKIP;
      else
            return TRAIL_RELOC;
    } else
      return TRAIL_IND_RELOC;
  }
};

extern void gccontrol_pass2();

enum {
  SHOULD_COPY,
  SHOULD_MARK,
  SHOULD_CHECK_MARK,
  SHOULD_RELOC,
  SHOULD_NEITHER,
  SHOULD_LEAVE
  };

extern void mark_compact();
extern void fast_copy();
extern struct rusage gc_rusage;
extern int getrusage(...);
extern CellPtr H_entry_value;
extern CellPtr H2_entry_value;
extern CellPtr TR_entry_value;
extern CellPtr TR2_entry_value;

extern void gc_init();
extern void global_sweep();
extern void restore_cps();
extern ChoiceRecord SAVED_CP;
extern void gctrail_pass11();
extern void compute_stats();
```

```
/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

#ifdef WITH_GC
#include <stream.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "tags.h"
#include "instr.h"
#include "hash_table.h"
#include "string_table.h"
#include "scan.h"
#include "inst_args.h"
#include "inst_table.h"
#include "memory.h"
#include "basics.h"
#include "top_level.h"
#include "gc.h"
#include "mark_copy.h"

/* LOCAL DECLARATIONS */

static DownStack FAST_MARK_STACK;
static CopyStack FAST_COPY_STACK;

/* if does not point directly to new space, either it dereferences to */
/* a pointer to new space that belongs to some living environment, */
/* that will be traced later on, or to some old environment, which */
/* modification would then have been trailed. Therefore, there is no */
/* need to dereference */
void Env::fast_copy()
{
#ifdef WITH_VIRTUAL_BACK
  Cell* y = e + Y1_ENV_OFFSET + already_treated;
  Cell* y0 = e + Y1_ENV_OFFSET + size;
  for (; y < y0; y++) {
    Cell* ptr = y;
    Cell val = *ptr;
    while (get_tag(val) == TAGREF && addr(val) >= E0 && addr(val) != ptr) {
      ptr = addr(val);
      val = *ptr;
    }
    if (get_tag(val) == TAGCONST) continue;
    if (to_new_space(addr(val)))
      copy_from_base(ptr);
  }
#else
  Cell* y = e + Y1_ENV_OFFSET + already_treated;
  Cell* y0 = e + Y1_ENV_OFFSET + size;
  for (; y < y0; y++) {
    if (get_tag(*y) == TAGCONST) continue;
    if (to_new_space(addr(*y)))
            copy_from_base(y);
  }
#endif
}

/* Needs to make sure that no unbound variable is left in registers */
void fast_copy_restore_top_env()
{
  Cell* PreviousE = cellp(E[E_ENV_OFFSET]);
  int arity = instrp(E[P_ENV_OFFSET])->arg2;
  E = PreviousE;
  for (int i = 0; i < arity; i++) {
```

```
    X[i] = deref(E[Y1_ENV_OFFSET + i]);
    if (X[i] == make_ptr(TAGREF, &E[Y1_ENV_OFFSET + i])) {
     Cell new_var = make_ptr(TAGREF, FAST_COPY_STACK.top());
     FAST_COPY_STACK.push(new_var);
     X[i] = E[Y1_ENV_OFFSET + i] = new_var;
    }
   }
  }

  /* CHOICE POINTS */

  void setup_cps_fast_copy()
  {
   /* treat the case of the cps such that B.h == HMIN now */
   Cell* b = B2 = B;
   while (cellp(b[H_CP_OFFSET]) == HMIN) {
    b[H_CP_OFFSET] = cell(H2);
    b += FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
   }

   /* creates a topmost choice point */
   B -= FIXED_CP_SIZE;
   B[E_CP_OFFSET] = cell(E);
   B[H_CP_OFFSET] = cell(H);
   B[TR_CP_OFFSET] = cell(TR);
   B[P_CP_OFFSET] = 0; /* unused */
   B[SIZE_CP_OFFSET] = 0;

   /* set B2 to be above E2, TR2 as well; save previous contents */
   SAVED_CP.tr = cellp(B2[TR_CP_OFFSET]);
   SAVED_CP.e = cellp(B2[E_CP_OFFSET]);
   SAVED_CP.h = cellp(B2[H_CP_OFFSET]);
   TR2 = min(TR2, SAVED_CP.tr);
   E2 = max(E2, SAVED_CP.e);
   B2[TR_CP_OFFSET] = cell(TR2);
   B2[E_CP_OFFSET] = cell(E2);
   B2[H_CP_OFFSET] = cell(HMIN);
  }

  void restore_cps_fast_copy()
  {
   /* restore B2 to its initial contents */
   B2[TR_CP_OFFSET] = cell(SAVED_CP.tr);
   B2[E_CP_OFFSET] = cell(SAVED_CP.e);
   B2[H_CP_OFFSET] = cell(SAVED_CP.h);

   /* compute the new values of H, H2, TR, TR2, E2 */
   H = HMIN;
   H2 = FAST_COPY_STACK.top();
   TR = TR2 = cellp(B[TR_CP_OFFSET]);
   E2 = E;

   /* remove the dummy topmost choice point */
   B += FIXED_CP_SIZE;
  }

   /* takes advantage of the fact that the tag bit is in the lower bits */
  void fast_copy_trail_1()
  {
   register Cell* tr0 = cellp(B[TR_CP_OFFSET]);
   register Cell* tr = cellp(B2[TR_CP_OFFSET]);
   register Cell* copy_tr = tr;
   register Cell* h = cellp(B2[H_CP_OFFSET]);
   register Cell* e = cellp(B2[E_CP_OFFSET]);
```

```
 for (; tr > tr0; tr--) {
  if (cellp(*tr) < h || (cellp(*tr) < e && cellp(*tr) >= E0))
    *copy_tr-- = *tr;
 }
 B[TR_CP_OFFSET] = cell(copy_tr);
}

void fast_copy_trail_2()
{
 register Cell* tr0 = cellp(B[TR_CP_OFFSET]);
 register Cell* tr = cellp(B2[TR_CP_OFFSET]);
 for (; tr > tr0; tr--) {
  register Cell* ptr = addr(*tr);
  if (ptr >= E2 || (ptr < E0 && ptr >= HMIN))
    continue;
  if (pointer_to_new(*ptr))
    copy_from_base(ptr);
 }
}

void fast_copy_trail()
{
 fast_copy_trail_1();
 fast_copy_trail_2();
}

 /* control stacks */

 /* we do the traversal of the environment stack and the choice point */
 /* stack together. that way we can avoid having to traverse the */
 /* records twice, and we do not have to use marking nor any extra */
 /* space: just two extra structures. */
 /* will be quite easy to add virtual backtracking inside this routine */
 /* it works as follows: first visit all envs above the topmost choice */
 /* point. then visit all envs that are above the next living env. two */
 /* loops alternating, one visiting next living envs, one visiting the */
 /* next preserved envs. if a given env is shared, its living part is */
 /* first entirely marked, then we wait until the last choice point */
 /* that preserved that env and mark the part that is preserved. */

void fast_copy_control()
{
 /* only living objects in that case */
  Env env(E);
 for (;;) {
  if (env.e <= E2) {
    if (env.e == E2)
          env.fast_copy();
    break;
   }
   env.fast_copy();
   env.next();
 }
}

 /* we save and later restore the topmost entry in the COPY_STACK at */
 /* the time this routine is called. This is to simplify the algorithm */
 /* and avoid copying many times. Here, the main difficulty is the */
 /* correct treatment of refs. Since the order does not matter any */
 /* more here, we can be even a bit more efficient. Each time we */
 /* encounter a ref, we dereference it. If we get a constant, we just */
 /* copy the constant into the origin. If we get an unbound variable, */
 /* we rebind it backwards, and set the original pointer to unbound. */
 /* This may create pointers from new to base space for a while, so we */
```

```
/* should be careful. The idea is to always dereference fully, no */
/* matter what, and look at where the result is. Only stop when */
/* marked. Using the FAST_MARK_STACK helps a lot, though it cannot be */
/* deeper than one element. */

void copy_from_base(Cell* p)
{
 FAST_MARK_STACK.init(B);
 FAST_MARK_STACK.push(p);
 for (;;) {
  Cell* var;
  if (FAST_COPY_STACK.nonempty())
   var = FAST_COPY_STACK.pop();
  else if (FAST_MARK_STACK.nonempty())
   var = FAST_MARK_STACK.pop();
  else
   break;

  switch (get_tag(*var)) {
  case TAGCONST:
   break;
  case TAGREF:
   {
        Cell* ptr = addr(*var);
        if (ptr < HMIN || ptr >= E0) {
         if (*var == *ptr) {
          if (ptr > var)
           *ptr = *var = make_ptr(TAGREF, var);
         } else {
          *var = *ptr;
          FAST_MARK_STACK.push(var);
         }
        } else if (marked(ptr)) {
         *var = make_ptr(TAGREF, reloc_addr(ptr));
        } else if (*var == *ptr) {
         *ptr = *var = make_ptr(TAGREF, var);
        } else {
         *var = *ptr;
         FAST_MARK_STACK.push(var);
        }
   }
   break;
  case TAGLIST:
   {
        Cell* list = addr(*var);
        if (list >= HMIN) {
         if (marked(list)) {
          *var = make_ptr(TAGLIST, reloc_addr(list));
         } else {
          *var = make_ptr(TAGLIST, FAST_COPY_STACK.top());
          for (int i = 0; i < 2; i++) {
           mark(list + i);
           Cell* dest = FAST_COPY_STACK.top();
           FAST_COPY_STACK.push(list[i]);
           set_reloc_addr(list + i, dest);
          }
         }
        }
   }
   break;
  case TAGSTRUCT:
   {
        Cell* str = addr(*var);
        if (str >= HMIN) {
```

```
            if (marked(str)) {
              *var = make_ptr(TAGSTRUCT, reloc_addr(str));
            } else {
              *var = make_ptr(TAGSTRUCT, FAST_COPY_STACK.top());
              int i0 = get_int(str[1]) + 2;
              for (int i = 0; i < i0; i++) {
                mark(str + i);
                Cell* dest = FAST_COPY_STACK.top();
                FAST_COPY_STACK.push(str[i]);
                set_reloc_addr(str + i, dest);
              }
            }
          }
        }
        break;
      }
    }
  }

  /* Basic Initializations */
  void fast_copy_gc_init()
  {
  #ifdef WITH_VIRTUAL_BACK
    MARK = 2 * ((GC_COUNTER % 127) + 1); /* values from 2 to 254 */
  #else
    MARK = (GC_COUNTER % 255) + 1;       /* values from 1 to 255 */
  #endif
    GC_COUNTER++;
    FAST_COPY_STACK.init(H2);
  }

  /* Collect some data */
  /* some basic data: mark(scan,recovered), copy(scan,recovered), cputime */
  /* the data are given in number of cells, milliseconds. */

  void fast_copy_stats()
  {
    gc_scanned += H_entry_value - HMIN;
    gc_copy_scanned += H_entry_value - HMIN;
    gc_survivors += H2 - H2_entry_value;
    tr_scanned += TR2_entry_value - TR_entry_value;
    tr_survivors += TR2_entry_value - TR;
    if (DISPLAY_GC) {
      cout << "gc (";
      display_stat1("copy", H_entry_value - HMIN, H2 - H2_entry_value);
      display_stat1("tr", TR2_entry_value-TR_entry_value, TR2_entry_value-TR);
    }
  }

  /* top level */
  /* assumes that GC_DOES_COPY. Should also work if everything is above */
  /* the topmost choice point, though slower than the special purpose */
  /* fast_copy garbage collector */

  void fast_copy()
  {
    init_stats();
    store_regs_in_env();
    setup_cps_fast_copy();
    fast_copy_gc_init();
    init_marking_table();
    fast_copy_trail();
    fast_copy_control();
    fast_copy_restore_top_env();
```

```
  restore_cps_fast_copy();
  fast_copy_stats();
  compute_stats();
}

#endif
```

```
/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

#ifdef WITH_GC
#include <stream.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "tags.h"
#include "instr.h"
#include "hash_table.h"
#include "string_table.h"
#include "scan.h"
#include "inst_args.h"
#include "inst_table.h"
#include "memory.h"
#include "basics.h"
#include "top_level.h"
#include "gc.h"
#include "mark_copy.h"

/* ENVS */

void Env::mark_sweep()
{
#ifdef WITH_VIRTUAL_BACK
  Cell* y = e + Y1_ENV_OFFSET + already_treated;
  Cell* y0 = e + Y1_ENV_OFFSET + size;
  for (; y < y0; y++) {
    Cell* ptr = y;
    Cell val = *ptr;
    while (get_tag(val) == TAGREF && addr(val) >= E2 && addr(val) != ptr) {
      ptr = addr(val);
      val = *ptr;
    }
    if (get_tag(val) == TAGCONST) continue;
    if (to_new_space(addr(val)))
      mark_from_base_sweep(ptr);
  }
#else
  Cell* y = e + Y1_ENV_OFFSET + already_treated;
  Cell* y0 = e + Y1_ENV_OFFSET + size;
  for (; y < y0; y++) {
    if (get_tag(*y) == TAGCONST) continue;
    if (to_new_space(addr(*y)))
            mark_from_base_sweep(y);
  }
#endif
}

/* CHOICE POINTS */

/* creates a choice point at the top that is above everything else. */
/* It is easier to code gctrail that way: don't have to worry about */
/* boundary conditions any more.  */
void setup_cps_pass1_sweep()
{
/* creates a topmost choice point */
  B -= FIXED_CP_SIZE;
  B[E_CP_OFFSET] = cell(E);
  B[H_CP_OFFSET] = cell(H);
  B[TR_CP_OFFSET] = cell(TR);
  B[P_CP_OFFSET] = 0; /* unused */
  B[SIZE_CP_OFFSET] = 0;
```

```
/* find BMIDDLE and B2 */
 Cell* b = B;
 while (cellp(b[H_CP_OFFSET]) > HMIN) {
  b += FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
 }
 B2 = b;

/* treat the case of the cps under B2 such that B.h == HMIN now */
 while (cellp(b[H_CP_OFFSET]) == HMIN) {
  b[H_CP_OFFSET] = cell(H2);
  b += FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
 }

/* set B2 to be above TR2 as well; save previous contents */
 SAVED_CP.tr = cellp(B2[TR_CP_OFFSET]);
 SAVED_CP.e = cellp(B2[E_CP_OFFSET]);
 SAVED_CP.h = cellp(B2[H_CP_OFFSET]);
 TR2 = min(TR2, SAVED_CP.tr);
 E2 = max(E2, SAVED_CP.e);
 B2[TR_CP_OFFSET] = cell(TR2);
 B2[E_CP_OFFSET] = cell(E2);
 B2[H_CP_OFFSET] = cell(H2);

/* just to limit the modifications with the mark_copy case */
 HMIDDLE = HMIN;
}

/* just replace unmarked2 by unmarked */
void Choice::virtual_backtrack_sweep()
{
#ifdef WITH_VIRTUAL_BACK
 Cell* var0 = tr;
 Cell* var = cellp(b[TR_CP_OFFSET]);
 tr = var;
 for (; var > var0; var--) {
  Cell* ptr = addr(*var);
  if (ptr >= E0) {
   Cell val = *ptr;
   while (get_tag(val) == TAGREF && addr(val) >= E0 && addr(val) != ptr) {
        ptr = addr(val);
        val = *ptr;
   }
   if (pointer_to_new(*ptr) && unmarked(addr(*ptr)))
        *addr(*var) = *var;
  } else if (ptr >= HMIN) {
   if (unmarked(ptr))
        *ptr = *var;
  }
 }
#endif
}

void Choice::mark_sweep()
{
#ifdef WITH_VIRTUAL_BACK
 virtual_backtrack_sweep();
 Cell* x = b + X1_CP_OFFSET;
 Cell* x0 = x + b[SIZE_CP_OFFSET];
 for (; x < x0; x++) {
  Cell* ptr = x;
  Cell val = *ptr;
  while (get_tag(val) == TAGREF && addr(val) >= E2 && addr(val) != ptr) {
   ptr = addr(val);
   val = *ptr;
```

```
    }
    if (get_tag(val) == TAGCONST) continue;
    if (to_new_space(addr(val)))
            mark_from_base_sweep(ptr);
  }
#else
  Cell* x = b + X1_CP_OFFSET;
  Cell* x0 = x + b[SIZE_CP_OFFSET];
  for (; x < x0; x++) {
    if (get_tag(*x) == TAGCONST) continue;
    if (to_new_space(addr(*x)))
      mark_from_base_sweep(x);
  }
#endif
}

/* THE TRAIL STACK */

void gctrail_pass12_sweep()
{
  register Cell* tr0 = cellp(B[TR_CP_OFFSET]);
  register Cell* tr = cellp(B2[TR_CP_OFFSET]);
  for (; tr > tr0; tr--) {
    register Cell* ptr = addr(*tr);
    if (ptr >= E2 || (ptr < E0 && ptr >= HMIN))
      continue;
    if (pointer_to_new(*ptr))
      mark_from_base_sweep(ptr);
  }
}

void gctrail_pass1_sweep()
{
  gctrail_pass11();
  gctrail_pass12_sweep();
}

void gctrail_pass2_sweep()
{
  TrailCP cp(B2, B);
  Cell* tr0 = cp.tr;
  Cell* tr = cp.tr;
  Cell* copy_tr = cp.tr;
  while (cp.nonempty()) {
    tr = tr0;
    tr0 = cp.next_tr;
    for (; tr > tr0; --tr) {
      Cell* ptr = addr(*tr);
      switch (cp.pass2_action_sweep(ptr)) {
      case TRAIL_SKIP:
              break;
      case TRAIL_RELOC:
              *copy_tr-- = relocate(TAGREF, ptr);
              break;
      case TRAIL_IND_RELOC:
              *ptr = check_and_relocate(*ptr);
              *copy_tr-- = *tr;
              break;
      }
    }
    cp.update_tr(copy_tr);
    cp.next();
  }
}
```

```
/* control stacks */

/* just replace marking by sweep marking */

void gccontrol_pass1_sweep()
{
 /* first, take care of living cells */
  Env env(E);
  for (;;) {
    if (env.e <= E2) {
      if (env.e == E2)
            env.mark_sweep();
      break;
    }
    env.mark_sweep();
    env.next();
  }

 /* now, take care of preserved cells */
  Choice cp(E, B);
  for (;;) {
    if (cp.last()) break;
    cp.mark_sweep();
    cp.mark_preserved_envs_sweep();
    cp.next();
  }
}

 /* simple, sweep marking */

 /* suppose p is a global stack pointer; can't point to env stack */
 /* should be recoded to use a table lookup instead of all those tests */

static DownStack MARK_SWEEP_STACK;
void mark_from_base_sweep(Cell* p)
{
  MARK_SWEEP_STACK.init(B);
  MARK_SWEEP_STACK.push(p);
  for (;;) {
    Cell* var;
    if (MARK_SWEEP_STACK.nonempty())
      var = MARK_SWEEP_STACK.pop();
    else
      break;

    switch (get_tag(*var)) {
    case TAGCONST:
      break;
    case TAGREF:
      {
            Cell* ptr = addr(*var);
            if (ptr >= HMIN && unmarked(ptr)) {
              mark(ptr);
              MARK_SWEEP_STACK.push(ptr);
            }
      }
      break;
    case TAGLIST:
      {
            Cell* list = addr(*var);
            if (list >= HMIN) {
              for (int i = 0; i < 2; i++) {
                if (unmarked(list + i)) {
```

```
                mark(list + i);
                MARK_SWEEP_STACK.push(list + i);
              }
            }
          }
    }
    break;
  case TAGSTRUCT:
    {
          Cell* str = addr(*var);
          if (str >= HMIN && unmarked(str)) {
            int i0 = get_int(str[1]) + 2;
            for (int i = 0; i < 2; i++)
              mark(str + i);
            for (i = 2; i < i0; i++) {
              mark(str + i);
              MARK_SWEEP_STACK.push(str + i);
            }
          }
    }
    break;
  }
 }
}

void mark_compact_stats()
{
 gc_scanned += H_entry_value - HMIN;
 gc_survivors += H2 - H2_entry_value;
 tr_scanned += TR2_entry_value - TR_entry_value;
 tr_survivors += TR2_entry_value - TR;
 if (DISPLAY_GC) {
  cout << "gc (";
  display_stat1("global", H_entry_value - HMIN, H2 - H2_entry_value);
  display_stat1("trail",TR2_entry_value-TR_entry_value, TR2_entry_value-TR);
 }
}

void init_marking_table_sweep()
{
#ifdef WITH_VIRTUAL_BACK
 if (MARK != 2) return;
#else
 if (MARK != 1) return;
#endif
 register int* p = (int*) MKMIN;
 register int* p0 = HMIN;
 while (p < p0)
  *p++ = 0;
}

 /* basic initializations */

 /* need to initialize MARK2 in case this is used with mark_copy */
void gc_init_sweep()
{
#ifdef WITH_VIRTUAL_BACK
 MARK = 2 * ((GC_COUNTER % 127) + 1); /* values from 2 to 254 */
#else
 MARK = (GC_COUNTER % 255) + 1;      /* values from 1 to 255 */
#endif
 GC_COUNTER++;
}
```

```
/* top level */
/* assumes that GC_DOES_COPY. Should also work if everything is above */
/* the topmost choice point, though slower than the special purpose */
/* fast_copy garbage collector */

void mark_compact()
{
  init_stats();
  store_regs_in_env();
  setup_cps_pass1_sweep();
  gc_init_sweep();
  init_marking_table_sweep();
  cp_to_cp_forward();
  gctrail_pass1_sweep();
  cp_to_cp_backward();
  gccontrol_pass1_sweep();
  global_sweep();
  gccontrol_pass2();
  cp_to_cp_forward();
  gctrail_pass2_sweep();
  cp_to_cp_backward();
  restore_top_env();
  restore_cps();
  mark_compact_stats();
  compute_stats();
}

#endif
```

```
/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

#ifdef WITH_GC
#include <stream.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "tags.h"
#include "instr.h"
#include "hash_table.h"
#include "string_table.h"
#include "scan.h"
#include "inst_args.h"
#include "inst_table.h"
#include "memory.h"
#include "basics.h"
#include "top_level.h"
#include "gc.h"
#include "mark_copy.h"

/* LOCAL DECLARATIONS */

/* choice point that separates copying from marking */
CellPtr BMIDDLE, HMIDDLE, B2;

/* various stacks used during marking */
static DownStack MARK_STACK;
static UpStack REF_STACK;
static CopyStack COPY_STACK;

/* incremented at each GC. Just a counter */
int GC_COUNTER;

/* the mark used for marking. Equals to GC_COUNTER modulo 255 + 1 */
unsigned char MARK;
#ifdef WITH_VIRTUAL_BACK
static unsigned char MARK2;

inline void mark2(Cell* p)
{ MKMIN[p - HMIN] = (marked(p)) ? MARK : MARK2; }

inline int marked2(Cell* p)
{ return (marked(p) || MKMIN[p - HMIN] == MARK2); }

inline int unmarked2(Cell* p)
{ return (unmarked(p) && MKMIN[p - HMIN] != MARK2); }

#endif

/* ENVIRONMENTS and REGISTERS */
/* creates a new environment at the top of the stack, and saves the */
/* registers in it. Then put yet another one above it, with nothing */
/* in it. Easier to restore than adding the registers to the current */
/* environment. */
static Instr dummy_instr;
void store_regs_in_env()
{
  int arity = instr_args[ARG_PROC]->get_arity(P->arg1);
  arity = (NUMBER_OF_REGISTERS < arity) ? NUMBER_OF_REGISTERS : arity;
  dummy_instr.arg2 = arity;
  for (int i = 0; i < arity; i++)
    E[Y1_ENV_OFFSET + i] = X[i];
  Cell* NewE = E + arity + E_TOP_OFFSET;
  NewE[B_ENV_OFFSET] = 0; /* unused */
```

```
  NewE[E_ENV_OFFSET] = cell(E);
  NewE[P_ENV_OFFSET] = cell(&dummy_instr);
  E = NewE;
}

 /* restore the top of the stack as before the call to store_regs_in_env */
void restore_top_env()
{
  Cell* PreviousE = cellp(E[E_ENV_OFFSET]);
  int arity = instrp(E[P_ENV_OFFSET])->arg2;
  E = PreviousE;
  for (int i = 0; i < arity; i++)
    X[i] = E[Y1_ENV_OFFSET + i];
}

 /* if does not point directly to new space, either it dereferences to */
 /* a pointer to new space that belongs to some living environment, */
 /* that will be traced later on, or to some old environment, which */
 /* modification would then have been trailed. Therefore, there is no */
 /* need to dereference */
void Env::mark()
{
#ifdef WITH_VIRTUAL_BACK
  Cell* y = e + Y1_ENV_OFFSET + already_treated;
  Cell* y0 = e + Y1_ENV_OFFSET + size;
  for (; y < y0; y++) {
    Cell* ptr = y;
    Cell val = *ptr;
    while (get_tag(val) == TAGREF && addr(val) >= E2 && addr(val) != ptr) {
      ptr = addr(val);
      val = *ptr;
    }
    if (get_tag(val) == TAGCONST) continue;
    if (to_new_space(addr(val)))
      mark_from_base(ptr);
  }
#else
  Cell* y = e + Y1_ENV_OFFSET + already_treated;
  Cell* y0 = e + Y1_ENV_OFFSET + size;
  for (; y < y0; y++) {
    if (get_tag(*y) == TAGCONST) continue;
    if (to_new_space(addr(*y)))
            mark_from_base(y);
  }
#endif
}

void Env::update()
{
  Cell* y = e + Y1_ENV_OFFSET + already_treated;
  Cell* y0 = e + Y1_ENV_OFFSET + size;
  for (; y < y0; y++)
    *y = check_and_relocate(*y);
}

 /* CHOICE POINTS */

ChoiceRecord SAVED_CP;

 /* if less than a threshold, use mark_compact instead */
const float COPY_THRESHOLD = 0.2;

int deterministic()
{
```

```
  return (cellp(B[H_CP_OFFSET]) <= HMIN);
}

int enough_to_copy()
{
  Cell* H_THRESHOLD = &HMIN[(int) ((float) (H-HMIN)*COPY_THRESHOLD)];
  Cell* b = B;
  while (cellp(b[H_CP_OFFSET]) > H_THRESHOLD)
    b += FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
  return (cellp(b[H_CP_OFFSET]) <= HMIN);
}

 /* creates a choice point at the top that is above everything else. */
 /* It is easier to code gctrail that way: don't have to worry about */
 /* boundary conditions any more.  */
void setup_cps_pass1()
{
 /* creates a topmost choice point */
  B -= FIXED_CP_SIZE;
  B[E_CP_OFFSET] = cell(E);
  B[H_CP_OFFSET] = cell(H);
  B[TR_CP_OFFSET] = cell(TR);
  B[P_CP_OFFSET] = 0; /* unused */
  B[SIZE_CP_OFFSET] = 0;

 /* find BMIDDLE and B2 */
  BMIDDLE = B;
  Cell* b = B;
  while (cellp(b[H_CP_OFFSET]) > HMIN) {
    BMIDDLE = b;
    b += FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
  }
  B2 = b;

 /* treat the case of the cps under B2 such that B.h == HMIN now */
  while (cellp(b[H_CP_OFFSET]) == HMIN) {
    b[H_CP_OFFSET] = cell(H2);
    b += FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
  }

 /* set B2 to be above TR2 as well; save previous contents */
  SAVED_CP.tr = cellp(B2[TR_CP_OFFSET]);
  SAVED_CP.e = cellp(B2[E_CP_OFFSET]);
  SAVED_CP.h = cellp(B2[H_CP_OFFSET]);
  TR2 = min(TR2, SAVED_CP.tr);
  E2 = max(E2, SAVED_CP.e);
  B2[TR_CP_OFFSET] = cell(TR2);
  B2[E_CP_OFFSET] = cell(E2);
  B2[H_CP_OFFSET] = cell(HMIN);

 /* cache the H entry of BMIDDLE in a global variable */
  HMIDDLE = cellp(BMIDDLE[H_CP_OFFSET]);
}

void setup_cps_pass2()
{
 /* restore B2 to its initial contents */
  B2[TR_CP_OFFSET] = cell(SAVED_CP.tr);
  B2[E_CP_OFFSET] = cell(SAVED_CP.e);
  B2[H_CP_OFFSET] = cell(SAVED_CP.h);

 /* take BMIDDLE as B2: copied stuff appears as old form now on */
  B2 = BMIDDLE;
  H2 = COPY_STACK.top();
```

```
 B2[H_CP_OFFSET] = cell(H2);

 /* set B2 to be above TR2 as well; save previous contents */
 SAVED_CP.tr = cellp(B2[TR_CP_OFFSET]);
 SAVED_CP.e = cellp(B2[E_CP_OFFSET]);
 TR2 = min(TR2, SAVED_CP.tr);
 E2 = max(E2, SAVED_CP.e);
 B2[TR_CP_OFFSET] = cell(TR2);
 B2[E_CP_OFFSET] = cell(E2);
}

void restore_cps()
{
 /* restore B2 to its initial contents */
 B2[TR_CP_OFFSET] = cell(SAVED_CP.tr);
 B2[E_CP_OFFSET] = cell(SAVED_CP.e);

 /* relocate the H entries to their correct, final position */
 Cell* b = B;
 while (b < B2) {
  b[H_CP_OFFSET] = cell(reloc_addr(cellp(b[H_CP_OFFSET])));
  b += FIXED_CP_SIZE + b[SIZE_CP_OFFSET];
 }

 /* compute the new values of H, H2, TR, TR2, E2 */
 H = HMIN;
 H2 = cellp(B[H_CP_OFFSET]);
 TR = TR2 = cellp(B[TR_CP_OFFSET]);
 E2 = E;

 /* remove the dummy topmost choice point */
 B += FIXED_CP_SIZE;
}

 /* hard to get all the benefit from this. The main problem is that we */
 /* cannot mark env variables as easily. Since this is applied only */
 /* after gctrail and gcenv, if an env variable is found to be */
 /* pointing to a location that is unmarked2 in new space, we know we */
 /* can reset it. We could extend that by dereferencing the var. If */
 /* the first entry to new space is not marked, we can reset the var */
void Choice::virtual_backtrack()
{
#ifdef WITH_VIRTUAL_BACK
 Cell* var0 = tr;
 Cell* var = cellp(b[TR_CP_OFFSET]);
 tr = var;
 for (; var > var0; var--) {
  Cell* ptr = addr(*var);
  if (ptr >= E0) {
   Cell val = *ptr;
   while (get_tag(val) == TAGREF && addr(val) >= E0 && addr(val) != ptr) {
        ptr = addr(val);
        val = *ptr;
   }
   if (pointer_to_new(*ptr) && unmarked2(addr(*ptr)))
        *addr(*var) = *var;
  } else if (ptr >= HMIN) {
   if (unmarked2(ptr))
        *ptr = *var;
  }
 }
#endif
}
```

```
/* This stack exactly simulates what would happen on backtracking */
/* supposing we encounter an infinite sequence of fails. This is */
/* really virtual backtracking! The problem is really the difference */
/* in sizes of the environments, depending on the point of view! That */
/* is the only reason why we need a stack (or marking bits). Stacks */
/* are preferable in general because they are faster and cleaner. */
/* the cost is on choice points only */

Choice::Choice(Cell* E, Cell* B)
{
 b = B;
 tr = cellp(B[TR_CP_OFFSET]);
 preserved.init(cellp(b[E_CP_OFFSET]));
 already_done.init(E);
}

void Choice::mark()
{
#ifdef WITH_VIRTUAL_BACK
 virtual_backtrack();
 Cell* x = b + X1_CP_OFFSET;
 Cell* x0 = x + b[SIZE_CP_OFFSET];
 for (; x < x0; x++) {
  Cell* ptr = x;
  Cell val = *ptr;
  while (get_tag(val) == TAGREF && addr(val) >= E2 && addr(val) != ptr) {
   ptr = addr(val);
   val = *ptr;
  }
  if (get_tag(val) == TAGCONST) continue;
  if (to_new_space(addr(val)))
        mark_from_base(ptr);
 }
#else
 Cell* x = b + X1_CP_OFFSET;
 Cell* x0 = x + b[SIZE_CP_OFFSET];
 for (; x < x0; x++) {
  if (get_tag(*x) == TAGCONST) continue;
  if (to_new_space(addr(*x)))
   mark_from_base(x);
 }
#endif
}

#define use(ACTION,PROC_NAME)\
void Choice::PROC_NAME()\
{\
 while (already_done.e > preserved.e)\
  already_done.next();\
 Env e_limit = already_done;\
 already_done = preserved;\
 while (preserved.e >= E2) {\
  if (preserved.e > e_limit.e) {\
   preserved.ACTION();\
   preserved.next();\
  } else if (preserved.e == e_limit.e) {\
   preserved.treated(e_limit.size);\
   preserved.ACTION();\
   break;\
  } else {\
   top_level_error("Inconsistent Path thru Env Stack");\
  }\
 }\
}
```

```
use(mark,mark_preserved_envs)
use(mark_sweep,mark_preserved_envs_sweep)
use(update,update_preserved_envs)
#undef use

void Choice::update()
{
  Cell* x = b + X1_CP_OFFSET;
  Cell* x0 = x + b[SIZE_CP_OFFSET];
  for (; x < x0; x++)
    *x = check_and_relocate(*x);
}

 /* rotates the size fields of the choice points [B2,B] down, putting */
 /* the one for B2 in B[SIZE_CP_OFFSET] */
void cp_to_cp_forward()
{
  int b2_size = B2[SIZE_CP_OFFSET];
  Cell* b = B;
  int size = b[SIZE_CP_OFFSET];
  while (b < B2) {
    b += FIXED_CP_SIZE + size;
    int temp = size;
    size = b[SIZE_CP_OFFSET];
    b[SIZE_CP_OFFSET] = temp;
  }
  B[SIZE_CP_OFFSET] = b2_size;
}

 /* do the opposite. composing those two should be a noop */
void cp_to_cp_backward()
{
  int b_size = B[SIZE_CP_OFFSET];
  Cell* b = B2;
  int size = b[SIZE_CP_OFFSET];
  while (b > B) {
    b -= FIXED_CP_SIZE + size;
    int temp = size;
    size = b[SIZE_CP_OFFSET];
    b[SIZE_CP_OFFSET] = temp;
  }
  B2[SIZE_CP_OFFSET] = b_size;
}

 /* THE TRAIL STACK */

/* OLD VERSION
void gctrail_pass1()
{
  TrailCP cp(B2, B);
  register Cell* tr0 = cp.tr;
  register Cell* tr = cp.tr;
  register Cell* copy_tr = cp.tr;
  while (cp.nonempty()) {
    tr = tr0;
    tr0 = cp.next_tr;
    for (; tr > tr0; tr--) {
      register Cell* ptr = addr(*tr);
      switch (cp.pass1_action(ptr)) {
      case TRAIL_MARK:
              mark_from_base(ptr);
              *copy_tr-- = *tr;
              break;
      case TRAIL_KEEP:
```

```
            *copy_tr-- = *tr;
            break;
      case TRAIL_SKIP:
            break;
      }
    }
    cp.update_tr(copy_tr);
    cp.next();
  }
}
*/

/* takes advantage of the fact that the tag bit is in the lower bits */
void gctrail_pass11()
{
  TrailCP cp(B2, B);
  register Cell* tr0 = cp.tr;
  register Cell* tr = cp.tr;
  register Cell* copy_tr = cp.tr;
  while (cp.nonempty()) {
    tr = tr0;
    tr0 = cp.next_tr;
    Cell* e = cp.e;
    Cell* h = cp.h;
    for (; tr > tr0; tr--) {
      if (cellp(*tr) < h || (cellp(*tr) < e && cellp(*tr) >= E0))
            *copy_tr-- = *tr;
    }
    cp.update_tr(copy_tr);
    cp.next();
  }
}

void gctrail_pass12()
{
  register Cell* tr0 = cellp(B[TR_CP_OFFSET]);
  register Cell* tr = cellp(B2[TR_CP_OFFSET]);
  for (; tr > tr0; tr--) {
    register Cell* ptr = addr(*tr);
    if (ptr >= E2 || (ptr < E0 && ptr >= HMIN))
      continue;
    if (pointer_to_new(*ptr))
      mark_from_base(ptr);
  }
}

void gctrail_pass1()
{
  gctrail_pass11();
  gctrail_pass12();
}

/* B2 has been set to BMIDDLE meanwhile; only look at the top part of */
/* the trail above BMIDDLE now. */
/* Also, there is the special case of trail entries pointing to the */
/* part that has been copied. Some of those need relocation */
void gctrail_pass2()
{
  TrailCP cp(B2, B);
  Cell* tr0 = cp.tr;
  Cell* tr = cp.tr;
  Cell* copy_tr = cp.tr;
  while (cp.nonempty()) {
    tr = tr0;
```

```
    tr0 = cp.next_tr;
    for (; tr > tr0; --tr) {
      Cell* ptr = addr(*tr);
      switch (cp.pass2_action(ptr)) {
      case TRAIL_SKIP:
            break;
      case TRAIL_RELOC:
            *copy_tr-- = relocate(TAGREF, ptr);
            break;
      case TRAIL_COPY_RELOC:
            *copy_tr-- = relocate(TAGREF, ptr);
            ptr = reloc_addr(ptr);
            *ptr = check_and_relocate(*ptr);
            break;
      case TRAIL_IND_RELOC:
            *ptr = check_and_relocate(*ptr);
            *copy_tr-- = *tr;
            break;
      }
    }
    cp.update_tr(copy_tr);
    cp.next();
  }
}

/* control stacks */

/* we do the traversal of the environment stack and the choice point */
/* stack together. that way we can avoid having to traverse the */
/* records twice, and we do not have to use marking nor any extra */
/* space: just two extra structures. */
/* will be quite easy to add virtual backtracking inside this routine */
/* it works as follows: first visit all envs above the topmost choice */
/* point. then visit all envs that are above the next living env. two */
/* loops alternating, one visiting next living envs, one visiting the */
/* next preserved envs. if a given env is shared, its living part is */
/* first entirely marked, then we wait until the last choice point */
/* that preserved that env and mark the part that is preserved. */
/* the update is simple macro substitution from the mark */

#define use(ACTION,PRESERVED_ACTION,PROC_NAME)\
void PROC_NAME()\
{\
/* first, take care of living cells */\
  Env env(E);\
  for (;;) {\
    if (env.e <= E2) {\
      if (env.e == E2)\
            env.ACTION();\
      break;\
    }\
    env.ACTION();\
    env.next();\
  }\
/* now, take care of preserved cells */\
  Choice cp(E, B);\
  for (;;) {\
    if (cp.last()) break;\
    cp.ACTION();\
    cp.PRESERVED_ACTION();\
    cp.next();\
  }\
}
use(mark,mark_preserved_envs,gccontrol_pass1)
```

```
use(update,update_preserved_envs,gccontrol_pass2)
#undef use

/* new space itself: compaction phase */

/* not too hard. just go thru new area and the marking area in */
/* parallel. each time i encounter something marked, copy it down */
/* in copy space. leave behind in each location the relocation */
/* address (untagged). */
/* needs a second scan to compute the final addresses. proportional */
/* to m+n in total */
/* Also, for being able to restore global stack pointers uniformly, */
/* we add one entry at the top to relocate the topmost choice point */
/* entry correctly */
/* This is also the place to gather statistics about the efficiency */
/* of the garbage collector */
static Cell* H2_copy_value;
static Cell* H_copy_value;
void global_sweep()
{
 register Cell* p = HMIDDLE; /* from lowest cp segment */
 register Cell* p0 = H;
 register unsigned char* m = &MKMIN[HMIDDLE - HMIN];
 register Cell* h = H2;
 H_copy_value = HMIDDLE;

 /* sweep pass. Should always write relocation addresses */
 for (; p < p0; p++, m++) {
  if (*m == MARK) {
   *h = *p;
   *p = cell(h);
   h++;
  } else {
   *p = cell(h);
  }
 }

 /* relocation info for the topmost choice point */
 *p = cell(h);

 /* relocate pointers to new space */
 p = H2_copy_value = H2;
 H2 = p0 = h;
 for (; p < p0; p++) {
  if (pointer_to_new(*p))
   *p = relocate(*p);
 }
}

 /* the REF stack: delayed copying of variables in copy space */
 /* objects in the stack should be pointers to locations containing ref */
 /* pointers to cp_down */
 /* if virtual backtracking, we cannot guarantee visiting only once */
void gcref_pass1()
{
 while (REF_STACK.nonempty()) {
  Cell* var = REF_STACK.pop();
  Cell* ptr = addr(*var);
#ifdef WITH_VIRTUAL_BACK
  if (! to_new_space(ptr)) continue;
#endif
  if (unmarked(ptr)) {
   mark(ptr);
   Cell val = *ptr;
```

```
    set_reloc_addr(ptr, COPY_STACK.top());
    COPY_STACK.push(val);
    if (get_tag(val) == TAGREF && addr(val) >= HMIN)
        REF_STACK.push(reloc_addr(ptr));
   }
   *var = make_ptr(TAGREF, reloc_addr(ptr));
  }
}

/* marking */

/* we pass a pointer to the cell containing the pointer to the object */
/* to mark. not necessary for marking, but necessary for copying. */
/* we use the space at the top of the choice point stack (between */
/* choice point stack and the environment stack) as the marking stack. */
/* we need to initialize the marking area at each gc. here, since we */
/* use one byte per mark, we can rotate the mark, and reduce the cost */
/* of initialization by 255. */

/* when copying, don't mark ref pointers nor what they point to. we */
/* will do it later. also trail pointers from copy area to new area */
/* to speed up relocation. */

/* suppose p is a global stack pointer; can't point to env stack */
/* should be recoded to use a table lookup instead of all those tests */

/* OLD VERSION
inline int copy_or_mark(Cell* p)
{
  if (p < HMIN)
    return SHOULD_NEITHER;
  else if (p < HMIDDLE)
    return (marked(p)) ? SHOULD_RELOC : SHOULD_COPY;
  else
    return (marked(p)) ? SHOULD_CHECK_MARK : SHOULD_MARK;
}
*/

int copy_or_mark_table[2][2] = {
  {SHOULD_MARK, SHOULD_CHECK_MARK},
  {SHOULD_COPY, SHOULD_RELOC}
};

inline int copy_or_mark(Cell* p)
{
  if (p >= HMIN)
    return copy_or_mark_table[(p < HMIDDLE)][marked(p)];
  else
    return SHOULD_NEITHER;
}

/* In the copy part, a list or a structure is marked iff any of its */
/* elements is. */
void mark_from_base(Cell* p)
{
  MARK_STACK.init(B);
  MARK_STACK.push(p);
  for (;;) {
    Cell* var;
    if (COPY_STACK.nonempty())
      var = COPY_STACK.pop();
    else if (MARK_STACK.nonempty())
      var = MARK_STACK.pop();
    else
```

```
    break;

  switch (get_tag(*var)) {
  case TAGCONST:
    break;
  case TAGREF:
    {
          Cell* ptr = addr(*var);
          switch (copy_or_mark(ptr)) {
          case SHOULD_MARK:
            mark(ptr);
            MARK_STACK.push(ptr);
            break;
          case SHOULD_RELOC:      /* ptr to marked copied location */
            *var = make_ptr(TAGREF, reloc_addr(ptr));
            break;
          case SHOULD_COPY:
            REF_STACK.push(var);
            for (;; var = ptr, ptr = addr(*ptr)) {
              /* here, ptr is always a pointer to low cp segment */
#ifdef WITH_VIRTUAL_BACK
              if (get_tag(*ptr) != TAGREF) {
                MARK_STACK.push(ptr);
                mark2(ptr);
                break;
              }
              if (ptr < HMIN || *var == *ptr || marked2(ptr))
                break;
              mark2(ptr);
#else
              if (get_tag(*ptr) != TAGREF) {
                MARK_STACK.push(ptr);
                break;
              }
              if (ptr < HMIN || marked(ptr) || *var == *ptr)
                break;
#endif
            }
            break;
          case SHOULD_CHECK_MARK:
          case SHOULD_NEITHER:
            break;
          }
    }
    break;
  case TAGLIST:
    {
          Cell* list = addr(*var);
          switch (copy_or_mark(list)) {
          case SHOULD_CHECK_MARK: /* marked(car) && unmarked(cdr) */
            if (unmarked(list + 1)) {
              mark(list + 1);
              MARK_STACK.push(list + 1);
            }
            break;
          case SHOULD_MARK:
            for (int i = 0; i < 2; i++) {
              mark(list + i);
              MARK_STACK.push(list + i);
            }
            break;
          case SHOULD_COPY:
            *var = make_ptr(TAGLIST, COPY_STACK.top());
            for (i = 0; i < 2; i++) {
```

```
            mark(list + i);
            Cell* dest = COPY_STACK.top();
            COPY_STACK.push(list[i]);
            set_reloc_addr(list + i, dest);
           }
           break;
          case SHOULD_RELOC:
           *var = make_ptr(TAGLIST, reloc_addr(list));
           break;
          case SHOULD_NEITHER:
           break;
          }
    }
    break;
   case TAGSTRUCT:
    {
          Cell* str = addr(*var);
          switch (copy_or_mark(str)) {
          case SHOULD_MARK:
           int i0 = get_int(str[1]) + 2;
           for (int i = 0; i < 2; i++)
            mark(str + i);
           for (i = 2; i < i0; i++) {
            mark(str + i);
            MARK_STACK.push(str + i);
           }
           break;
          case SHOULD_COPY:
           *var = make_ptr(TAGSTRUCT, COPY_STACK.top());
           i0 = get_int(str[1]) + 2;
           for (i = 0; i < i0; i++) {
            mark(str + i);
            Cell* dest = COPY_STACK.top();
            COPY_STACK.push(str[i]);
            set_reloc_addr(str + i, dest);
           }
           break;
          case SHOULD_RELOC:
           *var = make_ptr(TAGSTRUCT, reloc_addr(str));
           break;
          case SHOULD_CHECK_MARK:
          case SHOULD_NEITHER:
           break;
          }
    }
    break;
  }
 }
}

/* should allocate a fixed size region, just under new area. Needs */
/* only be initialized once with 0s. For the rest, We can just flip */
/* and use a global variable, say MARK. MARK is initialized to the */
/* current gc number modulo 255. When it overflows, the area is */
/* cleared again. During marking, only MARK is written in the byte */
/* corresponding to the word to be written. To be marked just means */
/* that this mark is being written. Only called when MARK is null */

void init_marking_table()
{
#ifdef WITH_VIRTUAL_BACK
 if (MARK != 2) return;
#else
 if (MARK != 1) return;
```

```
#endif
  register int* p = (int*) MKMIN;
  register int* p0 = HMIN;
  while (p < p0)
    *p++ = 0;
}

 /* basic initializations */

void gc_init()
{
#ifdef WITH_VIRTUAL_BACK
  MARK = 2 * ((GC_COUNTER % 127) + 1); /* values from 2 to 254 */
  MARK2 = MARK + 1;                    /* values from 3 to 255 */
#else
  MARK = (GC_COUNTER % 255) + 1;       /* values from 1 to 255 */
#endif
  GC_COUNTER++;
  REF_STACK.init(E);
  COPY_STACK.init(H2);
}

 /* some basic data: mark(scan,recovered), copy(scan,recovered), cputime */
 /* the data are given in number of cells, milliseconds. */
struct rusage gc_rusage;
Cell* H2_entry_value;
Cell* H_entry_value;
Cell* TR_entry_value;
Cell* TR2_entry_value;
void init_stats()
{
  getrusage(RUSAGE_SELF, &gc_rusage);
  H2_entry_value = H2;
  H_entry_value = H;
  TR_entry_value = TR;
  TR2_entry_value = TR2;
}

void display_stat1(char* legend, int before, int after)
{
  float percent = (before) ? ((float) after/before) * 100 : 0;
  printf("%s (%d, %d, %2.1f), ", legend, before, after, percent);
}

void display_stat2(char* legend, int tb, int cb, int ta, int ca)
{
  float percentb = (tb) ? ((float) cb/tb) * 100 : 0;
  float percenta = (ta) ? ((float) ca/ta) * 100 : 0;
  printf("%s (%2.1f, %2.1f), ", legend, percentb, percenta);
}

int gc_scanned;
int gc_copy_scanned;
int gc_survivors;
int tr_scanned;
int tr_survivors;
float gc_time;

void compute_stats()
{
  struct timeval from = gc_rusage.ru_utime;
  getrusage(RUSAGE_SELF, &gc_rusage);
  struct timeval to = gc_rusage.ru_utime;
  float mstime = (float) to.tv_usec / 1000000 + to.tv_sec;
```

```
  mstime -= (float) from.tv_usec / 1000000 + from.tv_sec;
  gc_time += mstime;
  if (DISPLAY_GC)
    printf("time (%.3f) ).\n", mstime);
  if (trace_heap_flag)
    heap_usage.gc_enter(H_entry_value, H2_entry_value);
}

void mark_copy_stats()
{
  gc_scanned += H_entry_value - HMIN;
  gc_copy_scanned += H_copy_value - HMIN;
  gc_survivors += H2 - H2_entry_value;
  tr_scanned += TR2_entry_value - TR_entry_value;
  tr_survivors += TR2_entry_value - TR;
  if (DISPLAY_GC) {
    cout << "gc (";
    display_stat1("global", H_entry_value - HMIN, H2 - H2_entry_value);
    display_stat2("copy",
                  H_entry_value - HMIN, H_copy_value - HMIN,
                  H2 - H2_entry_value, H2_copy_value - H2_entry_value);
    display_stat1("tr", TR2_entry_value-TR_entry_value, TR2_entry_value-TR);
  }
}

/* top level */
/* assumes that GC_DOES_COPY. Should also work if everything is above */
/* the topmost choice point, though slower than the special purpose */
/* fast_copy garbage collector */

int DISPLAY_GC;

void mark_copy()
{
  init_stats();
  store_regs_in_env();
  setup_cps_pass1();
  gc_init();
  init_marking_table();
  cp_to_cp_forward();
  gctrail_pass1();
  cp_to_cp_backward();
  gccontrol_pass1();
  gcref_pass1();
  setup_cps_pass2();
  global_sweep();
  gccontrol_pass2();
  cp_to_cp_forward();
  gctrail_pass2();
  cp_to_cp_backward();
  restore_top_env();
  restore_cps();
  mark_copy_stats();
  compute_stats();
}

int WHICH_GC = MARK_COPY;
int CHECK_GC_LIMIT;
int GC_COUNT_LIMIT;

/* we optimize the mark_copy case. Clearly, if there is nothing to */
/* copy, we should rather use mark_compact. It is faster! Around 7% */
/* faster in the case of gccomp. */
void garbage_collector()
```

```
{
 if (CHECK_GC_LIMIT && GC_COUNTER >= GC_COUNT_LIMIT) {
  cerr << "GC Limit passed\n";
 }
 switch (WHICH_GC) {
 case MARK_COPY:
  mark_copy();
  break;
 case MARK_COPY_FAST_COPY:
  if (deterministic())
   fast_copy();
  else
   mark_copy();
  break;
 case MARK_THRESHOLD:
  if (enough_to_copy())
   mark_copy();
  else
   mark_compact();
  break;
 case MARK_COMPACT:
  mark_compact();
  break;
 case MARK_COMPACT_FAST_COPY:
  if (deterministic())
   fast_copy();
  else
   mark_compact();
  break;
 default:
  top_level_error("Select GC algorithm first\n");
  break;
 }
 if (TR - H2 <= HMAXHARD - HMIN) {
  top_level_error("Global Stack Overflow\n");
 }
}

void find_pointer(Cell val)
{
 Cell* p;
 Cell* p0;

#define use(FROM,TO,NAME)\
 for (p = FROM, p0 = TO; p < p0; p++) {\
  if (*p == val)\
   cerr << NAME << " [" << (p - FROM) << "] \n";\
 }
use(H0,H2,"H0")
use(HMIN,HMAXSOFT,"HMIN")
use(E0,E,"E0")
use(B,B0,"B")
use(TR,TR0,"TR")
#undef use
}

#endif
```